

RELIABLE WEB APPLICATIONS

Development of a Web Cube prototype

Master Thesis
Thesis No: INF / SCR-05-58

Institute of Information and Computing Sciences
Utrecht University

Ivaylo Gochkov

August 31, 2006

Abstract

Web Cube is a server-side abstract and formal programming model for writing web applications. It allows temporal properties critical for the safety of an application to be specified. This thesis describes a prototype implementation of Web Cube. It is implemented in the functional language Haskell. The chosen implementation approach is to translate Web Cube source and specifications to Haskell's component skeletons and WebFunctions code. An engine supporting automated testing of Web Cube specifications is also provided.

Contents

1	Introduction	5
1.1	Problem statement	5
1.2	Chosen approach	6
1.3	A quick look to <i>Web Cube</i>	6
1.4	Literature	7
1.5	Description of the remaining chapters	8
2	Prototype overview	9
2.1	Features	9
2.2	Architecture	9
2.3	Sessions and states	11
2.4	Reused software	11
3	A Hello World Example	12
3.1	Steps	15
3.2	Requirements	16
4	Web Cube	17
4.1	Seuss	17
4.2	<i>Web Cube</i> concept + architecture	18
4.3	<i>Web Cube</i> Application	20
4.4	<i>Web Cube</i> Contracts	21
5	WebVote application	24
5.1	WebVote	24
5.2	VoteServer contract	27
6	Existing software	31
6.1	WebFunctions	31
6.2	WASH/HTML	32
6.3	HaXml	33
6.4	QuickCheck	34
6.4.1	Properties	34

6.4.2	The class <i>Arbitrary</i>	35
6.4.3	Observing data	35
6.5	Haskell	36
6.5.1	Haskell records	36
6.5.2	Monads	37
6.5.3	Concurrent Haskell	37
7	Testing	39
7.1	Testing of Refinement	39
7.2	Testing action	43
7.3	Testing Method	43
7.4	Invariant	44
7.5	Temporal properties	45
7.5.1	Testing with omega-Automaton	46
8	Generator	51
8.1	Generating WebFunctions	51
8.2	Generating component skeleton	53
8.3	Generating tests	56
9	Final remarks	59
9.1	Future work	59
9.2	Conclusion	59
A	<i>Web Cube</i> Grammar	60
B	<i>Web Cube</i> Keywords	63
C	<i>Web Cube</i> Prototype: Options	64
D	WebVote.cube	65
E	VoteServer.ctr	67
F	Definition of terms	69

Chapter 1

Introduction

1.1 Problem statement

Nowadays, Internet plays a major role in our everyday life. Web applications are widely used for e-commerce, banking, information systems, etc. Security, reliability and correctness of web pages become compulsory properties. Unfortunately, most technologies used in web-page building (e.g. ASP, PHP, and Java Servlets) do not provide sufficient abstraction, which makes correctness analysis difficult.

Some of the problems are caused by the programming method used. It is common to write web applications by mixing fragments of HTML code with application code without specifying any relation between a page and the inserted code. In this way, it is very difficult to predict the behavior of the application. Syntax and type errors are detected when the code in question is executed for the first time, for example by opening the web page that activates the code. Besides, the languages used in web page building nowadays are very hard to verify because of the complicated semantic model (if plain Java semantics is used) or even impossible to verify because no semantic model is available (e.g. in the case of ASP, JSP, PHP).

Web Cube [1] is a well-defined programming model for constructing interactive component-based web applications. It offers an alternative to servlets. It is more abstract in the way it deals with program composition and concurrency. It is based on a formal semantic, namely Prasetya et al's component-based extension [5] of Misra's Seuss [9]. The formalism allows the specification and verification of properties critical for the safety of an application.

Plain servlets' model of concurrency is based on threads, whereas Seuss is based on atomic actions, which is much more abstract than threads. It makes checking temporal properties much easier. It *basically* amounts to checking a set of state-predicates over the set of actions that constitute a *Web Cube* application, whereas in threads we have to check these predicates at each control point within each thread where interference may occur.

The latter will expose us to the underlying programming language, e.g. Java (with its complicated semantics) in the case of servlets. Efficient implementation of atomic actions exists, e.g. Harris et al’s *transactional memory* in the functional language Haskell [26].

The objective of this thesis project is to build a prototype of *Web Cube*, as a proof of principle of the *Web Cube* concept.

1.2 Chosen approach

We translate *Web Cube* applications to the functional language Haskell, in particular to a library called *WebFunctions*. This is a library by Robert van Herk [8] for writing web applications in Haskell. Haskell is chosen because we want to benefit from Haskell’s elegance and higher order feature as a functional language, and to reuse its wide range of libraries, e.g. *WebFunctions*, parser combinators, and QuickCheck — the latter facilitates automated testing.

Our prototype is also implemented in Haskell. It includes a *Web Cube* parser, a syntax validator, a *WebFunctions* generator for actual execution of *Web Cube* applications, a component skeleton generator, and a tests generator. Verification support is not included in the prototype — though the parser can be extended to generate, e.g. HOL fragments for verification. We do support validation via automated testing.

We also considered to implement the prototype by ”syntactically embedding” *Web Cube* into the theorem prover HOL [13]. Syntactical embedding (as opposed to semantical embedding [12]), allows us to reuse much of HOL’s own parser and type system. Embedding in HOL was proposed (rather than embedding in e.g. Stratego) because we wanted to access HOL’s rich libraries of verification utilities. Such an embedding (rather than implementing it from scratch) can minimize the implementation effort, especially since it is just intended to be a prototype. Unfortunately, this approach limits the syntax of *Web Cube* as we have to take into account all HOL’s own syntax. We decided not to pursue this path.

1.3 A quick look to *Web Cube*

The code below is how we can write a *HelloWorld* program in *Web Cube*:

```
application HelloWorld
{
  cube home
  {
    method void home()
```

```
{  
    respond("Hello World!")  
}  
}
```

The program defines a web application. The syntax is Java-like (influenced by its underlying Seuss syntax). An application is composed from the so-called cubes and resources. The HelloWorld application above has one cube called `home` and no resource. An entry point of a *Web Cube* application is the method 'home' of the cube 'home'. The cubes specify how the application responds to the client's requests, whereas the resources are (interfaces to) state persistent reactive programs shared over multiple applications. An application may also have its own variables, which define its 'local' state. This state is maintained per user session.

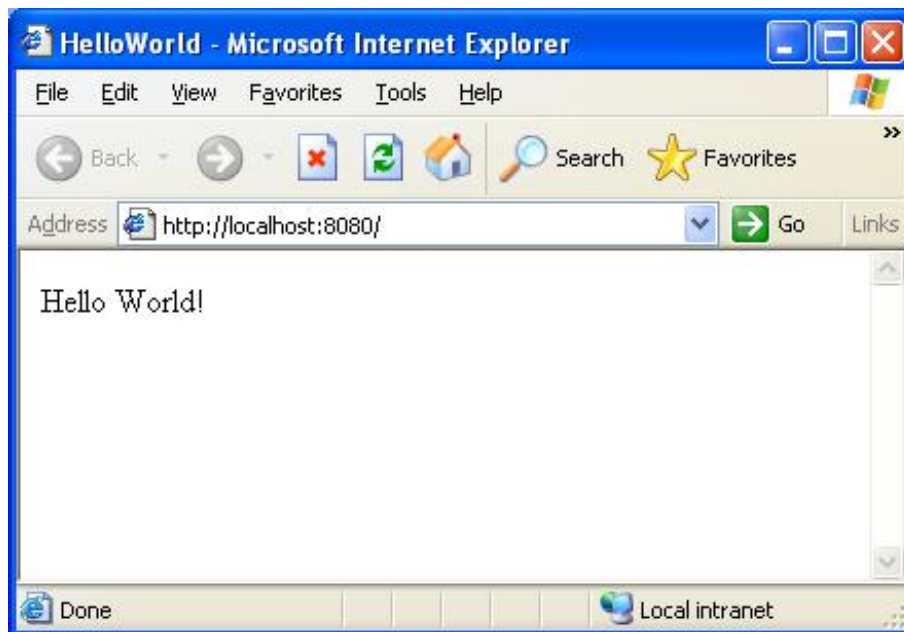


Figure 1.1: HelloWorld application: Web Page

The *Web Cube* prototype will compile this program via Haskell to an executable. Figure 1.1 shows a screen shot of a running HelloWorld application.

1.4 Literature

The work in this thesis is built upon a number of works, both theoretical and practical, by others. We will mention some which we think have a primary role in our implementation.

Web Cube is described in a paper by Prasetya, Vos, and Swierstra [2]. The paper describes *Web Cube* architecture, execution model, semantics, and a component-based logic. It does not specify any syntax (other than what can be gleaned from a few examples) — so, part of our implementation work is to propose a concrete syntax for *WebCube*. The semantics of *Web Cube* is given in terms of Seuss boxes. Seuss is a formal framework for modelling distributed systems. It is described in Misra’s ‘A Discipline of Multiprogramming’ [9]. Seuss itself is an outgrowth of a formal system called UNITY [3], which offers a very abstract view on concurrency.

Web Cube applications are implemented by translating them to *WebFunctions* applications. *WebFunctions* is described in Herk’s master thesis [8]. The implementation of resources requires supports for atomic computation over multiple shared variables. Fortunately, there is already such a support in Haskell, namely Harris et al’s STM library.

Our prototype supports automated testing of a resource against its specification in the form of a ‘contract’ (which may include a limited form of temporal properties). We use QuickCheck, a Haskell tool for automated random testing of predicates over values written by Claessen and Hughes [30]. In case of a failure, a counter-example is shown. The tool is commonly used to unit test Haskell functions against pre- and post-conditions. Our prototype transforms a contract to a set of predicates and test automata. These are checked in lock-step execution against the implementation of a web application. QuickCheck is used to randomly generate inputs and to facilitate predicate checking and the reporting of a violation.

1.5 Description of the remaining chapters

Chapter 2 gives an overview of the prototype’s features and architecture, as well as the reused software. Chapter 4 explains the concept and the *Web Cube* architecture. A detailed example of a *Web Cube* application is demonstrated in chapter 5. A brief description of existing software used in this thesis is given in chapter 6. The last three chapters explain in details the implementation of the prototype’s generator and the testing technique used.

Chapter 2

Prototype overview

This section gives an overview of the implemented prototype, its main features, architecture, the handling of sessions and states, and the reused software.

2.1 Features

The *Web Cube* prototype implements the concept of *Web Cube* applications. The prototype's main features are:

- it supports a component-based approach;
- it can generate a skeleton for a resource, based on a provided contract;
- it can generate an automated testing of a resource based on a provided contract;
- it supports session state and shared state over different sessions ¹.

2.2 Architecture

Figure 2.1 shows the architecture of the Web Cube prototype. The prototype takes *.cube or *.ctr files as inputs. The first ones are files describing *Web Cube* applications, and the second ones are contracts. Such file is first tokenized by the *Web Cube* scanner and then parsed by the *Web Cube* parser to produce an abstract syntax tree (AST). It further performs syntax validation, type checking (not implemented in this prototype), and code generation operate on the AST—the last proceeds only if syntax validation validates it positively.

There are three generators in the prototype: *WebFunctions* generator, component generator, and tests generator. The first generator turns a *Web Cube* application code to a

¹Shared state over different applications is not implemented because of the *WebFunctions* limitation to run one application in a time.

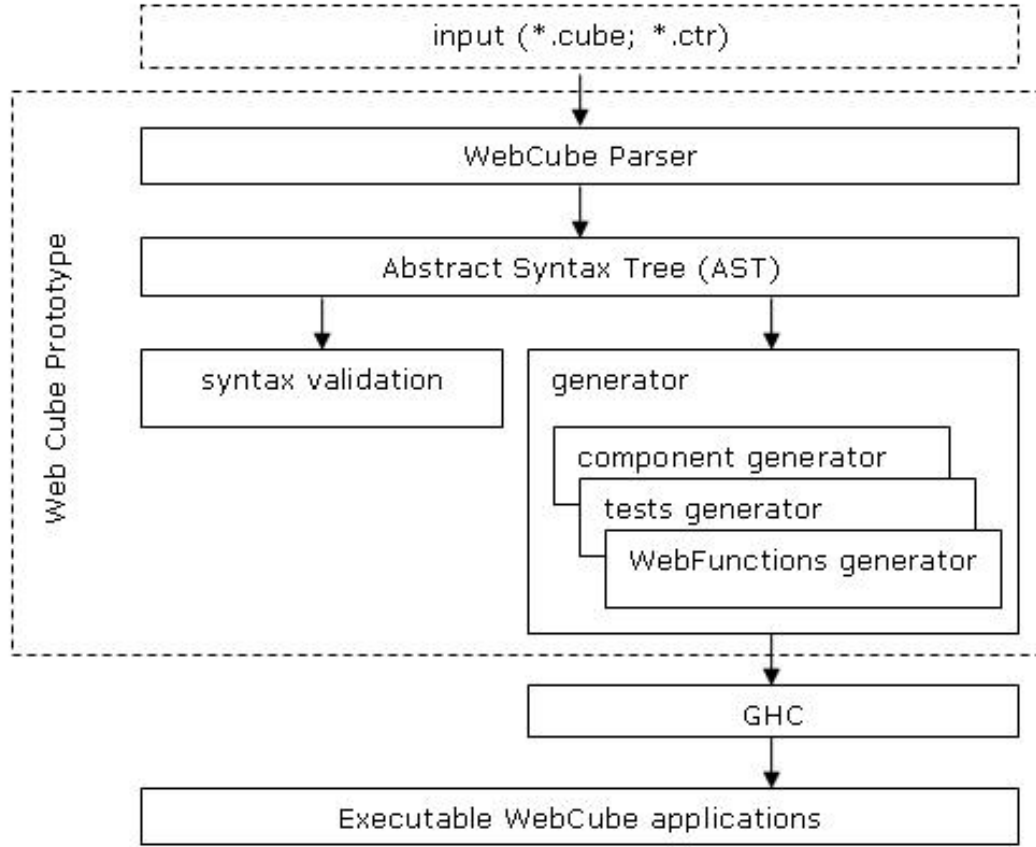


Figure 2.1: WebCube Prototype Architecture

runnable Haskell code. This Haskell code relies on the *WebFunctions* library to build an actual web application. Right after the generation, the prototype also provides the possibility for automatic compiling with the help of the GHC compiler.

The component generator can generate a skeleton of a resource from a contract. This is just a tool to make it easier for the programmer to implement a resource, if it is not available yet. A skeleton is not a complete code, so the programmer still has to fill in the missing details.

The tests generator takes a contract of a resource as its input, and using the information found in the contract, produces a Haskell code that will test properties promised in the contract against the resource. The testing is automatic, and it is implemented with the help of the QuickCheck tool.

2.3 Sessions and states

Sessions are handled internally by *WebFunctions*. That is, session management is hidden from the programmer; so, he needs not to think of how to bring the right session to the right user.

In *Web Cube* application there is a hierarchy of states. The state of an application consists of the states of its resources and those of its cubes. A resource's state is shared over different sessions, whereas the cube's state is not. The state of a cube consists of the states of its variables and the states of its methods. A method's state contains variables from the HTML response, which need to be transferred with the page session—for example the values from "input" tags.

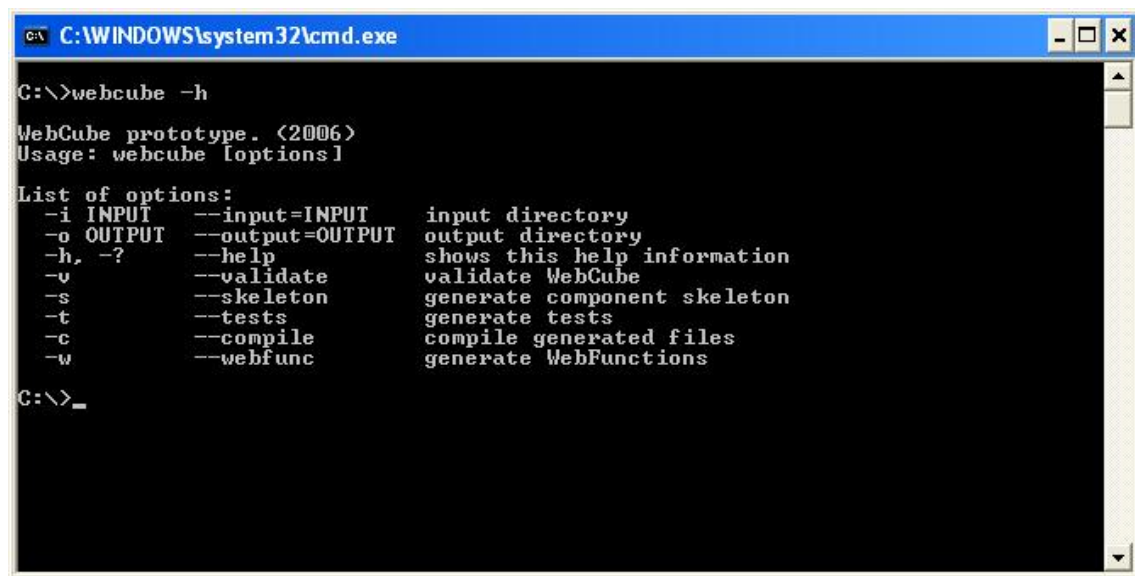
2.4 Reused software

During the implementation we try to reuse existing software as much as possible. The prototype is implemented in Haskell. The *Web Cube* tokenizer, parser and pretty printer use the UUST library. The HTML fragments in the *Web Cube* **respond** are parsed with the help of HaXML—see also Section 6.3. In order to run a *Web Cube* application, it is translated to a Haskell code. This code makes use of the *WebFunctions* library—see also Section 6.1. The QuickCheck tool is used to implement automated testing of contracts—see also Section 6.4.

Chapter 3

A Hello World Example

The *Web Cube* prototype is a tool implemented in Haskell. The main purpose of the tool is to parse Web Cube source files and to generate test properties and WebFunctions files. If the prototype is run with the `-h` option, it will show the list with possible options (figure 3.1)



```
C:\WINDOWS\system32\cmd.exe

C:\>webcube -h

WebCube prototype. <2006>
Usage: webcube [options]

List of options:
-i INPUT    --input=INPUT    input directory
-o OUTPUT    --output=OUTPUT  output directory
-h, -?      --help          shows this help information
-v          --validate       validate WebCube
-s          --skeleton       generate component skeleton
-t          --tests          generate tests
-c          --compile        compile generated files
-w          --webfunc        generate WebFunctions

C:\>_
```

Figure 3.1: Web Cube Prototype: Options

If no option is specified, the tool runs with default options which are `-vstw`. In this case the current directory is assumed to be an input and an output directory. A run of the tool with default options for WebVote application is shown in figure 3.2.

Chapter 1 shows a simple HelloWorld application which responds to user requests with a static "Hello World!" text. We can save the code to a file, e.g. `HelloWorld.cube` and put it e.g. in `C : \WebCube\HelloWorld\Source`. We can turn it to a Haskell code with:

```

C:\WINDOWS\system32\cmd.exe
C:\WebVote>webcube
Parsing WebCube .....OK
Validating WebCube
- checkHome.....OK
- checkRespond.....OK
- checkResources.....OK
Generating components .....OK
> C:\WebVote\VoteServer.hs
Generating tests .....OK
> C:\WebVote\VoteServerTest.hs
Generating WebFunctions ...OK
> C:\WebVote\WebVote.hs
C:\WebVote>

```

Figure 3.2: Web Cube Prototype: Default run

```
webcube -vcw -i C:\WebCube\HelloWorld\Source -o C:\WebCube\HelloWorld
```

The generated Haskell code can be compiled with GHC and produces an executable file, e.g. HelloWorld.exe. In the command used above, the 'v' option turns on syntax validation, 'w' generates the Haskell code, and 'c' turns on the subsequent compilation of the Haskell code to an executable. The 'i' option is used to specify the input directory where .cube files are searched; the 'o' options is used to specify the output directory. More information about options can be found in Appendix C.

To run the application, start the HelloWorld.exe file. The application will listen to the port 8080 for requests. A user using a web browser can visit it, e.g. via the URL <http://localhost:8080/> and he will see in his browser what is seen in Figure 1.1.

Below we show a slightly more sophisticated HelloWorld example. In particular, it maintains a state in the form of a variable called `counter` (in contrast, the previous HelloWorld is stateless). After a user starts a session, each time he requests the HelloWorld page, the `counter` will be incremented by one and its new value will be sent back to the user (which is then shown by his browser) together with the message "Hello World!".

```

application HelloWorld
{
  cube home
  {
    int counter = 0;

```

```

method void home()
{
    respond("<form method=post action='address.home.home'>
        Hello World!
        <p>Number of visits: <expr> counter </expr></p>
        <input type=submit value=next />
    </form>");

    counter <- counter + 1
}
}
}

```

Figure 3.3 shows what the user will see in his browser, if he interacts with the new HelloWorld example.

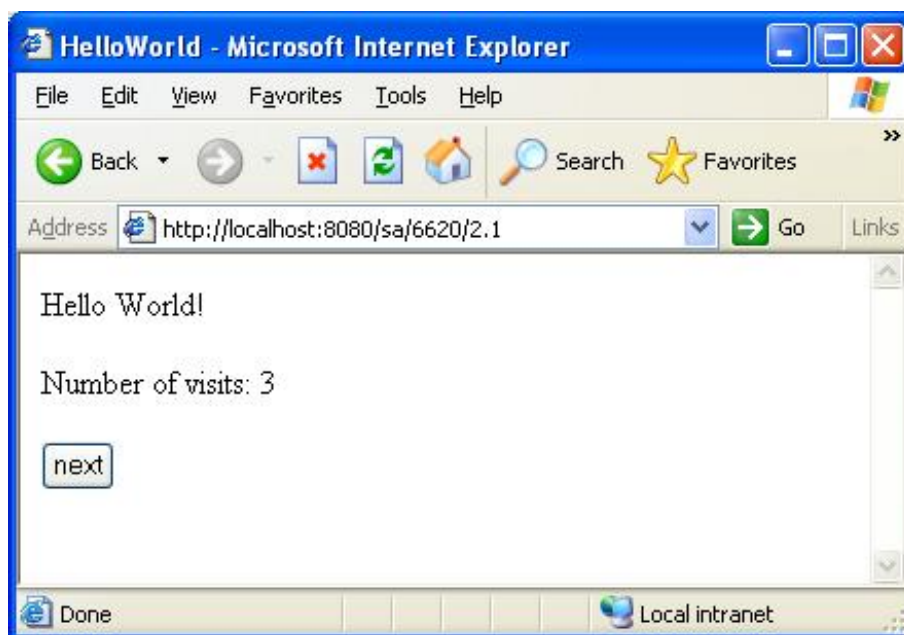


Figure 3.3: HelloWorld application with state: Web Page

The HTML sent back to the user is produced by the **respond** command, as shown in the code of HelloWorld above. If the reader takes a close look at the above HTML fragments, he will see two specialized constructs, unusual for HTML. The first one is the value of the attribute **action** (within the tag **form**). Instead of a URL, a qualified identifier is used to specify a local call to the method **home** of the cube **home**. This way after submission, the form will call the method and send the HTML responses produced by the method back to the user. The second construct is the tag **<expr> e </expr>**. The *Web Cube* generator will recognize it. It expects the *e* to be a *Web Cube* expression, which will be evaluated

from Haskell and the result is shown to the user.

3.1 Steps

The *Web Cube* prototype consists of the execution of seven steps.

1. Parsing *Web Cube* source files

The *Web Cube* prototype recognizes only two file extensions: *Web Cube* application files (.cube) and *Web Cube* contract files (.ctr). These files are parsed and an abstract syntax tree of the *Web Cube* type is made.

2. Validating the syntax of the parsed *Web Cube* application

The parser does not recognize all *Web Cube* specific constructs. That is why an additional validation of the syntax is necessary. This step checks if the parsed *Web Cube* application satisfies a predefined list with *Web Cube* rules. At this moment three major rules are checked:

- existence of a home cube and a method;
- existence of a respond in each method;
- each defined resource has a corresponding contract.

3. Generating a component skeleton

For each parsed contract, a component skeleton file is generated. This makes the programmer work easier later. He only has to extend this component skeleton to the real component implementation. The component skeleton contains all methods and actions defined in the contract, as well as the generated component state.

4. Generating test properties

For each contract in the *Web Cube* application a test file is generated. The file names are in the following format: *<contract name>Test.hs*

5. Compiling test properties

If the compile option 'c' is specified then the generated test files are also compiled with the help of GHC. The purpose of the test files is to test the implementation of the components. At the moment of generation, there is no ready implementation but just generated skeletons of the components, so eventual automatic run of the test is useless and is not included in the prototype. After implementing the components, the programmer can run the tests manually to verify his work.

6. Generating *WebFunctions* files

For each *Web Cube* application a working *WebFunctions* (ready to use) file is generated.

7. Compiling the generated *WebFunctions* files and the component skeleton
If the compile option 'c' is specified, then the generated *WebFunctions* files are also compiled.

3.2 Requirements

In order to be able to work with the *Web Cube* prototype, a few Haskell libraries and tools are required. The *Web Cube* parser uses UUST libraries (University of Utrecht, Software Technology) for parsing and pretty printing, as well as HaXML (section 6.3) for parsing the HTML in the responds. You could download a copy of UUST from <https://svn.cs.uu.nl:12443/repos/uust-repo/>. You will need a Glasgow Haskell Compiler to compile the generated Haskell files (ver. ≥ 6.5). A QuickCheck library (section 6.4) is required to run the generated component test files. And a *WebFunctions* is necessary (section 6.1) for compiling the generated *Web Cube* application files.

Chapter 4

Web Cube

4.1 Seuss

Seuss [9, 10] is a formal framework for modelling distributed systems and its main objective is to simplify multiprogramming. It is based on a simple but powerful mathematical system, and contains sufficient constructs of concurrent programming.

A system in Seuss is constructed from concurrent boxes. A box is a program with its own state defined by the variables of the box. A box can have a set of actions and methods. Both actions and methods can change the state of their own box or they can access methods of other boxes. Actions and methods can be guarded. Guarded actions (respectively, methods) are also called partial actions; otherwise they are total. A partial action blocks, if its guard is false; similarly, a partial method refuses a call, if its guard is false. An action may call methods, and a method may call other methods. A box is described in terms of a category (cat), and it is obtained by instantiating a cat. Cats and boxes are analogous to the class/object relation in Java.

Figure 4.1 shows an example of a Seuss system. The example defines two cats: *Counter* and *Say*. The system described consists of one box of the *Counter* category (*c*) and one box of the *Say* category (*s*). A box of the *Say* category has an action called Hello, which can change its own state and that of the box *c*.

Seuss extends the idea of UNITY [4] which is a simple formalism for describing and reasoning about distributed systems. As in UNITY, a Seuss system is executed by executing its actions. Each execution is infinite, where actions are executed in interleaving. At each step an action is selected non-deterministically and executed. If its guard is false, then the effect is just like a skip. Each execution has to be fair, in the sense that each action should not be ignored forever.

As in UNITY, Seuss defines three operators ***unless***, ***ensures*** and \mapsto (leads-to), which

```

program HelloWorld
  cat Counter
    var n: nat init 1
    total method Inc :: n := n + 1
  end { Counter }

  box c: Counter

  cat Say
    var text: string
    total action Hello ::
      text := "Hello world! " + c.n; c.Inc
  end { Say }

  box s: Say
end { HelloWorld }

```

Figure 4.1: Seuss program

we can use to describe the behaviour of a program. Given two state predicates p and q , a program P satisfies p **unless** q , if p holds during the execution of P at least until q holds. The program satisfies p **ensures** q , if satisfies p **unless** q and there is an action in P that establishes q . Let $P.meths$ is the set with methods and $P.acts$ is the set with actions in P :

Definition 4.1.1 : UNLESS

$$P \vdash p \text{ unless } q = (\forall x : x \in P.acts \bigcup P.meths : \{p \wedge \neg q\} x \{p \vee q\})$$

Definition 4.1.2 : ENSURES

$$P \vdash p \text{ ensures } q = (P \vdash p \text{ unless } q) \wedge (\exists a : a \in P.acts : \{p \wedge \neg q\} a \{q\})$$

While **ensures** specifies a progress guaranteed from a single action, \mapsto describes a progress in general. Program P satisfies $p \mapsto q$, if wherever p holds during the execution of P , q will also hold. The definition of \mapsto is more complicated. See [5] for details. We can check a broad range of temporal properties with the help of these three operators.

4.2 Web Cube concept + architecture

Figure 4.4 shows a simplified code of the HelloWorld application from Section 4.1. *Web Cube* is based on Seuss. More precisely, a *Web Cube* application such as the one in Figure

4.4 is basically just a Seuss box. So, we can use Seuss logic to check its temporal properties. All Seuss' features are inherited, e.g. because actions in Seuss are atomic, there is no need to program synchronization at the low level. So, in this sense, *Web Cube* is more abstract than a servlet. Furthermore *Web Cube* also uses Prasetya et al's component-based extension [5] of Seuss —we will say more on this in Section 4.4.

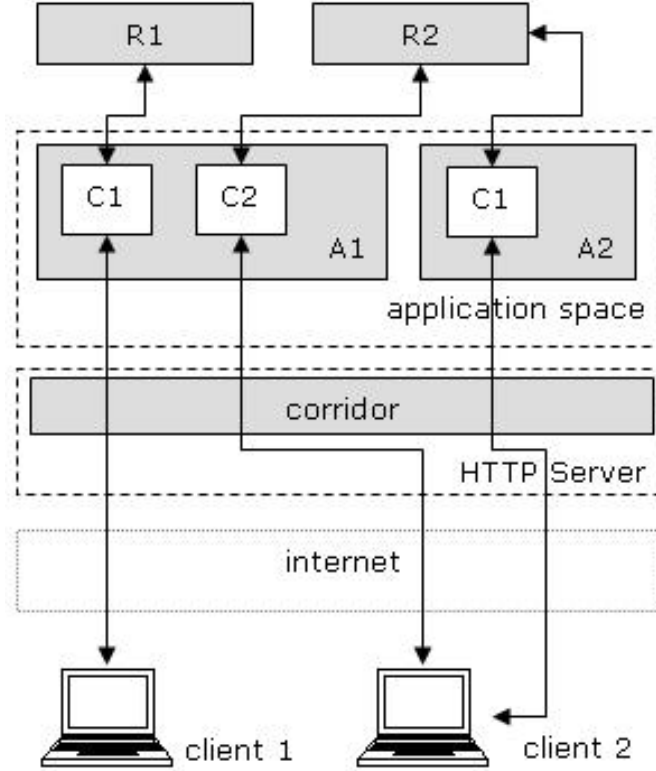


Figure 4.2: *Web Cube* Architecture

Figure 4.2 shows the architecture of the *Web Cube* execution system. *A1* and *A2* are *Web Cube* applications which can run on one or more machines. The imaginary space of all applications is called *application space*. The *Web Cube* enabled HTTP server has a special component called *corridor*. The role of the *corridor* is to redirect client requests to the right application and to collect back the respective responses. *R1* and *R2* are shared application resources. We can look at the *Web Cube* application as a set of abstract functionalities over collection of resources available to the client. The role of a cube is purely for serving client's requests. That is why a cube does not perform any action of its own. A resource, on the other hand, may execute actions to update its own state. The combination of a *Web Cube* application, its clients and the used resources form a distributed system.

A *cube* (e.g. *C1*, *C2*) is a server program waiting for client requests. The interaction is realised via HTTP connection. The user can use a browser to send HTTP requests to a cube. The cube can send back HTTP responses, which the user can view in his browser.

A method that produces HTML is called *writer* method. The life-time of a *cube* is only one session. A resource, on the other hand, is persistent. It represents a continuously running device (like a fax machine) or database system in an organization. A resource is shared across applications. As the resource can be a large and complicated system, *Web Cube* chooses to look at it as a black box. That is, *Web Cube* does not assume to know its full representation. A partial specification of how to interact with it and how it behaves is accepted. In terms of component-based approach, a resource is a component, and its partial specification is called contract.

Since the current prototype runs a *Web Cube* application on the same machine with the web server, *application space* and *corridor* are not implemented.

4.3 Web Cube Application

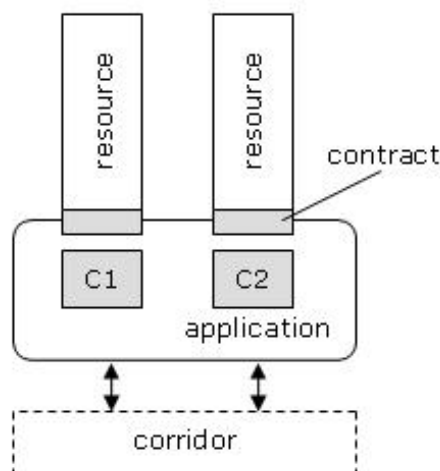


Figure 4.3: *Web Cube* Application Architecture

Figure 4.3 shows an architecture of a *Web Cube* application. A *Web Cube* application is built by composing *cubes* (*C1* and *C2*) and *resources*. Both the cubes and the resources are Seuss boxes with their own state.

The client can only interact with application's cubes; for safety reason he cannot interact directly with the resources. Note that *Web Cube* allows applications, wrapped as resources, to be composed to a new application which is a quite powerful feature.

Each *Web Cube* application has to have a *home* method placed in a *home* cube. This method is automatically called when an instance of the application is created. Like Java servlets, the *Web Cube* application responds printing HTML content using the *respond* method. *Web Cube* is not really at the same level as ASP or JSP as it does not support

```

application HelloWorld {
  cube home {
    n : Int = 0;

    writer method home() {
      respond("hello world! <@ n @>");
      n := n+1
    }
  }
}

```

Figure 4.4: *Web Cube* program

complicated code mixing as in the latter. In ASP and JSP code mixing is untyped, which is of course not a good practice of programming. There are techniques to allow typed code mixing, but this is beyond the scope of this project. *Web Cube* offers the possibility to embed program expressions into the response string using the symbols `<@` and `@>`. *Web Cube* has also another special form of inlining. Let m is a method name in cube with name c . In order to obtain the URI address of the method, we could write the following expression $address.c.m$. For example, a link to the home method can be made with:

```
respond("<a href=<@address.home.home@>>home</a>")
```

4.4 *Web Cube* Contracts

Since a resource can be a large and complicated system, it is represented as a black box specified by a contract. Such a contract includes a Seuss box that abstractly (and thus incompletely) specifies the resource behaviour as a reactive system. An application relies on the assumption that its resources behave according to their contracts. The responsibility for correct contracts is more likely to be given to the component developers, than to the application developer.

Essentially, a *Web Cube* application is just a Seuss system. Its properties can be expressed in terms of Seuss (temporal) operators. The original Seuss logic cannot however deal with black box components (the resources). Therefore *Web Cube* uses an extension [5]. The extension defines a notion of contract as a partial specification of a system, and extend the operators (*unless*, *ensures*, and \mapsto) accordingly. Of course since we now allow partial specifications, not all actual properties of a system can be verified.

In *Web Cube*, a contract consists of three parts: an *smodel*, an *invariant* and a *progress*. The *smodel* is a Seuss box which describes a reactive (non-terminating) program. It consists of a set of variables, a set of atomic guarded actions, and a set of methods. The invariant is a predicate which confines the set of states reachable by the program. The progress properties are defined with the help of the *leadsto* operator. A default contract pattern in *Web Cube* is:

```
contract VoteServer {
  smodel {
    ...
  }

  inv {
    ...
  }

  progress {
    ...
  }
}
```

The exemplary *Web Cube* application from Figure 4.4 can be viewed as a component for other *Web Cube* applications. This component implements simply a counter. A possible contract specifying it can look like this:

```
contract HelloWorld {
  int n = 0

  smodel {
    method void home()
    {
      n <- n + 1
    }
  }

  inv {
    n >= 0
  }

  progress {
    true leadsto (n > 0)
  }
}
```

The *smodel* specifies the behaviour of all methods and actions in the component — in this case only that of the **home** method. Calling the *home* method will increase the counter n . The invariant of the program is that n is always a positive number or zero. The progress property claims that n will always be a positive number.

Chapter 5

WebVote application

This example demonstrates the full range of *Web Cube* prototype features. The example consists of two parts: the first one is the *WebVote* application. We call the party that writes it the *application developer*. The second part describes a contract of a resource called *VoteServer*. It may be developed by a different party called the *component developer*. A complete definition of the *Web Cube* syntax can be found in Appendix A.

5.1 WebVote

The *WebVote* application is a small web application to a web based electronic voting. Its main web page shows a text box where the user types his vote, a submit button 'Vote' and two links: 'Vote info' and 'Stop voting'. This application uses a resource called *VoteServer* which is responsible for storing all submitted votes. The link 'Vote info' opens another web page displaying the number of valid votes counted so far. The link 'Stop voting' will stop the voting (after that the incoming votes will be ignored).

Let's develop *WebVote* step by step. With the keyword **application** followed by the application name, e.g. *WebVote* we define a new application. In the body of this application we can declare one or more cubes. There should always be a **home** cube, and it should have a **home** method, which plays the role of the application's main page.

```
application WebVote {  
  
    cube home  
    {  
        method void home()  
        {  
            ...  
        }  
    }  
}
```



```
    ...
}
```

We want to use the *VoteServer*, so we import it:

```
import VoteServer;
```

and then we can create an instance of this component using the keyword **resource** and bind a logical name to it, e.g. **r**:

```
resource r = VoteServer;
```

As already mentioned, the **home** method produces the *WebVote*'s main page. We want it to produce the submit form as shown in Figure 5.1. The form is produced by an HTML code which the method responds back to the client. This HTML is constructed inside the **respond** commands¹ of the method:

```
method void home()
{
    respond("<form method=post action='address.home.vote'>
        Enter your vote: <input type='text' name='v'>
        <input type=submit value='Vote'>
    </form>");
    respond("<p><a href='address.home.info'>Vote info</a></p>");
    respond("<p><a href='address.home.stop'>Stop voting</a></p>")
}
```

The prototype implements two types of code inlining in a slightly different manner than the original *Web Cube* concept. The first type, a call to local method, is placed directly into the values of the **href** and **action** attributes. The distinction between an URL and a local call is that the latter begins with an **address**; everything else is assumed to be an URL².

Now we have complete our main page. However, notice that this main page contains actions or hrefs like **address.home.vote**, which when activated will call the method **vote**

¹N.B. In the prototype, the body of a *respond* is parsed as a plain **String**. Because of restriction of the underlying library, you have to avoid any carriage return/linefeed characters within the *respond*, otherwise the parser will return error.

²So, we should not use the name **address** for a web page inside a *Web Cube* application, or always have to prefix a URL with e.g. **http://**

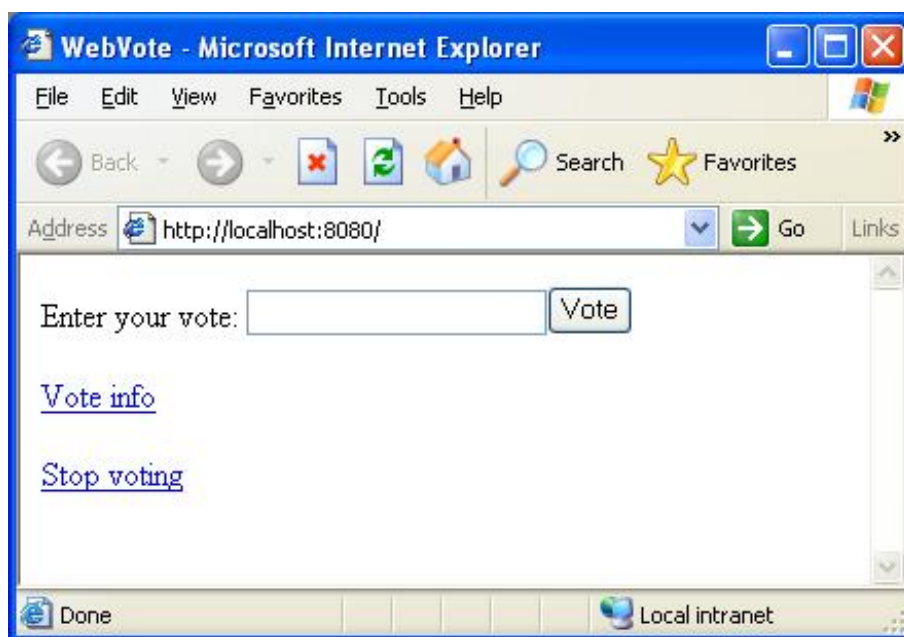


Figure 5.1: WebVote application: Home page

of the cube `home`. So we still need to implement this method. More precisely, the following methods still need to be implemented: `vote`, `info` and `stop`. The method `info` responds with the number of valid votes counted so far. The method `stop` will cause the resource `r` to stop accepting new votes. Internally, the resource maintains a boolean variable representing its 'open' or 'closed' state; `stop` will report the new value of this variable back to the user. The method `vote` is used to submit a new vote. Below we show the code of `vote` —that of the others can be found in the full listing of `WebVote.cube` in Appendix D.

```
method void vote(string v)
{
    if (r.vote(v)) then
        respond("<p>Your vote <b><expr> v </expr></b> was processed</p>")
    else
        respond("<p>Voting is closed.</p>");

    respond("<p align=center><a href='address.home.home'>Back</a></p>")
}
```

Notice the method above calls the method `vote` of the resource `r` (in the code `r.vote(v)`).

Above we can also see the second type of code inlining in *Web Cube*. Inside a `respond` we can tag an expression `e` like this: `<expr> e </expr>`. This will cause the expression to be first evaluated, and its result (turned to a string) is inserted as part of the HTML

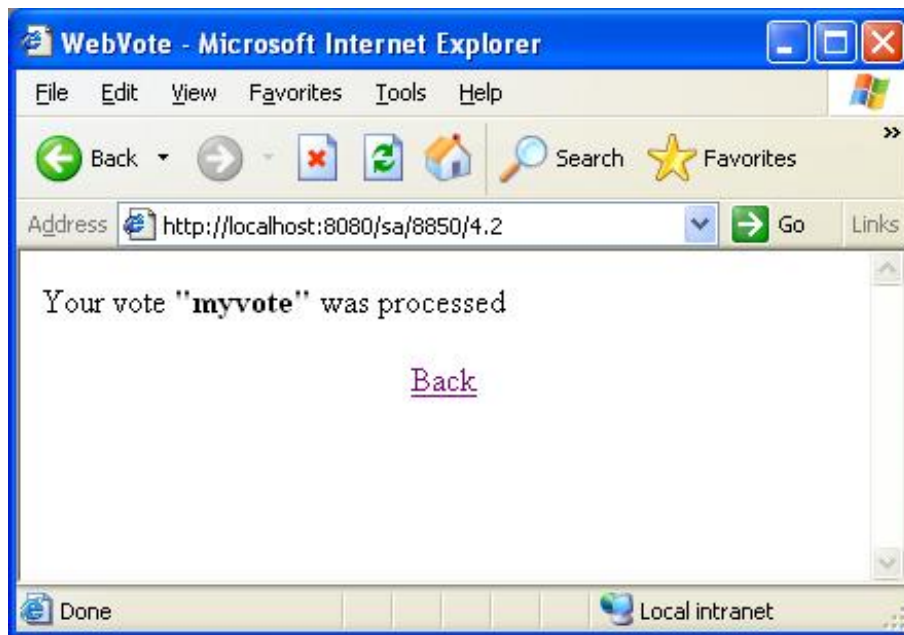


Figure 5.2: WebVote application: Vote page

response produced by `respond`.

In the current implementation every method is required to produce an HTML response. In a way, this makes a method comparable to a web page. This limitation can be dropped by producing a default response when a method does not produce one. We also do not implement the 'corridor'; so, there is no functionality which can collect responses from multiple methods.

5.2 VoteServer contract

This section shows and explains the contract of the `VoteServer` resource/component used in the `WebVote` application from the previous section.

A contract in *Web Cube* is an abstraction of a component. To write a reasonable and valid contract some knowledge about the described component is needed, e.g. we have to know how its methods and actions will behave. That is why this work is left to the component developer: he is responsible for providing a component that is consistent with its contract. The same developer may be also the developer of a *Web Cube* application that uses the component, but note that it does not have to be so.

Appendix E shows a possible contract of `VoteServer`. We will now explain step by step how this contract is constructed. Let us take the role of `VoteServer`'s developer.

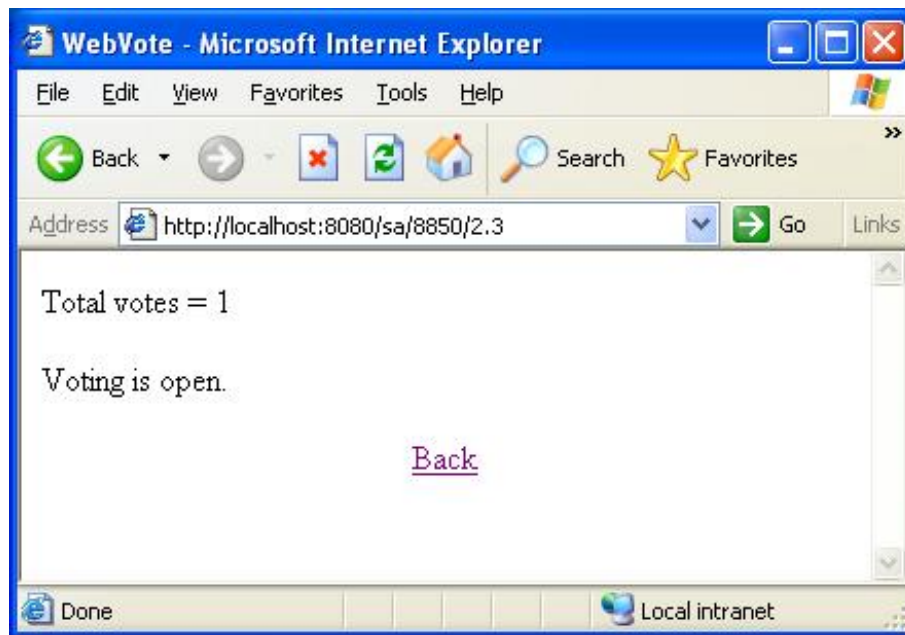


Figure 5.3: WebVote application: Info page

The component has to support the following functionalities: accepting incoming votes, validating votes, returning information about the number of counted votes, and closing the voting process. It maintains two variables: the set `in_` is used for storing the incoming votes, and the set `votes` for storing validated votes. It also maintains a boolean variable `open` indicating whether the voting process is open or closed. Internally, the component may have more variables, but knowledge about them is not needed to abstractly specify the component’s functionalities, as mentioned above.

The following is the declaration of the variables we need —for simplicity we assume that votes are just plain strings:

```
stringset in_   = \[\];
stringset votes = \[\];

bool open = true;
```

These variables are exposed in the contract.

Some of the required functionalities of `VoteServer` require interface with its user/environment, implemented via methods. These are the `vote`, `info`, and `stop` methods —their names indicate which functionalities they implement. Abstractly we can specify these methods as below; they are quite straightforward.

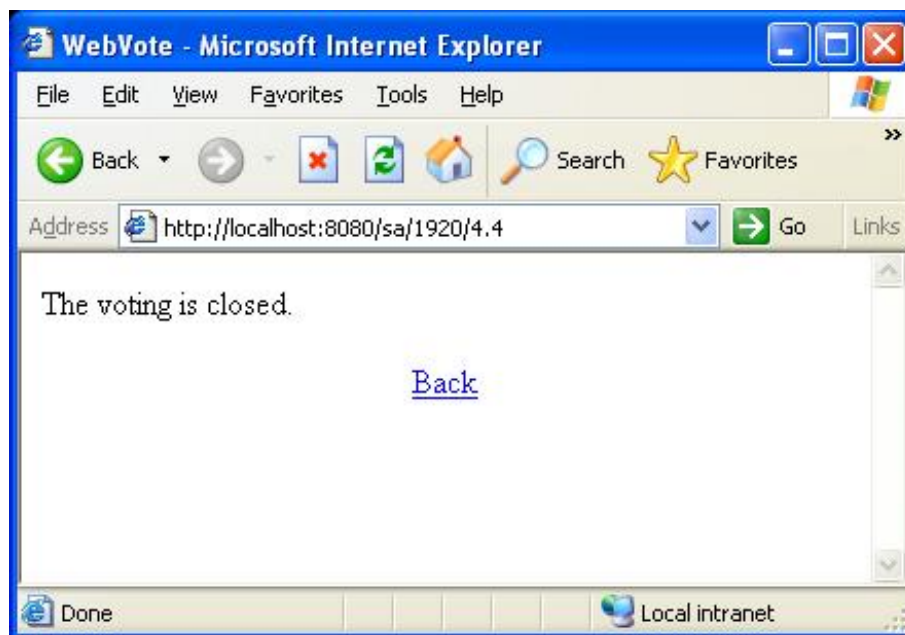


Figure 5.4: WebVote application: Stop page

```

method bool vote(string v)
{
    if (open) then {
        in_ <- insert(v)
    };

    return open
}

method int info()
{
    return size(votes)
}

method void stop()
{
    open <- false
}

```

There is one functionality left, namely that `VoteServer` has to validate the incoming votes. We decide to do this concurrently. That is, an incoming vote is not validated immediately. Instead, there are two actions that run concurrently: `move` and `validate`. The first moves the entire content of `in_` to some internal buffer. The second moves valid votes

from this internal buffer to `votes` and discards the rest. We decide that this internal buffer is not exposed in the contract. Abstractly, we can specify the behaviour of these actions as below. Note the specifications are only partial! This is because we decided to hide the internal buffer discussed above. So, the specification of e.g. `move` can specify that e.g. `in_` may be emptied. But it cannot specify where exactly in the internal buffer the votes are moved to, because the buffer is not exposed in the contract. Similarly, the specification of `validate` only says that it may extend `votes` with new votes.

```
action move {
    size(in_) > 0
    assert
    size(in_) == 0
}

action validate {
    true
    assert
    isSubsetOf((old votes), votes)
}
```

The contract so far is extended with the above specifications.

We expect that the set `votes` contains only valid votes all the time. This is an invariant property, which we can also specify in a contract. The notation is shown below:

```
inv {
    forall string v : (v in votes) : isValid(v)
}
```

Furthermore we require that any valid vote that comes in while the system is open for voting is guaranteed to eventually end up in the set `votes`. This is an example of a progress property, which we can also specify in a contract:

```
progress {
    forall string v : skip
    : ((v in in_) & isValid(v) & open) leadsto (v in votes)
}
```

If we now put all of the above specifications together, we obtain the full contract shown in Appendix E.

Chapter 6

Existing software

6.1 WebFunctions

WebFunctions is a framework for web site development implemented in Haskell. It is based on Apple's WebObjects, but it is type safe because of its implementation in Haskell. Some of the *WebFunctions* most important features are:

- *WebFunctions* applications have a loose coupling between model, view and controller code
- Transparent session state for the programmer
- Running inside a single process
- Easy reusability of the *WebFunctions* components
- Correctness of the generated HTML and SQL code

WebFunctions is type-safe and claim to generate correct HTML and SQL code. This claim is based on the fact that the errors are found by the compiler and the implementation of the underlying WASH/HTML is correct.

A *WebFunctions* application is created by implementing one or more components, controller code and a model. A component generates HTML code for the user's browser. The controller can be used to read/write a value in the model.

In the *WebFunctions* framework a session state is transparent to the programmer, meaning that the programmer does not need to think about binding the right session state to the right user, storing the session state after a response, etc. As a *WebFunctions* application runs inside one process, it is very easy to implement an application with a shared application state.

In *WebFunctions* every incoming HTTP request triggers a so-called *action* within the *WebFunctions* framework ¹. The evaluation of an action ultimately yields a response that is sent back to the client from whom the request originated. An action can have side effects, e.g. changes in an underlying database. There are four different kinds of actions: session actions that do access a session state and actions that do not access a session state (direct actions, the default action and error actions). The *WebFunctions* framework can automatically bind the right session state to the right session action. Error actions handle invalid requests. The default action is the entry point of a *WebFunctions* application. A direct action can access a state, but this state is global instead of session specific (used in shared states).

WebFunctions creates an action identifier from any correct URL. For example the following URL specifies a session action:

```
http://localhost:8080/sa/1023/1.2
```

WebFunctions will parse the full URL but the part before the third slash is ignored. Of importance for *WebFunctions* is `/sa/1023/1.2` where *sa* is session action, 1023 is the session identifier, 1 is context identifier and 2 is the component identifier. The generated URL for direct action is:

```
http://localhost:8080/da/DirectActionName
```

where *da* means that this is a direct action and *DirectActionName* is the name of the action. For direct actions the URL looks like this:

```
http://localhost:8080/
```

As mentioned earlier the direct action is the entry point of the application.

6.2 WASH/HTML

WASH/HTML ([28],[29]) is a Haskell DSEL(Domain specific embedded language) used in the implementation of *WebFunctions*. WASH/HTML is indirectly used in *Web Cube* prototype, namely in the *WebFunctions* generator. More details about *WebFunctions* and WASH/HTML are required in order to be able to understand the generated code from *Web Cube*.

WASH relies completely on monads (see 6.5.2). The monad transformer *WithHTML* is in the base of its HTML generator.

¹The meaning of an *action* in *WebFunctions* and *Web Cube* is different.


```

data WithHTML x m a = WithHTML { unWithHTML :: Element → m (a, Element) }

instance Monad m => Monad (WithHTML x m) where
  return a = WithHTML (λelem → return (a, elem))
  ma >>= f = WithHTML (λelem → unWithHTML ma elem >>= λ(a, elem') →
    unWithHTML (f a) elem')

```

The instance of *WithHTML* over $\gg=$ above, allows a sequential composition of WASH/HTML tags. For example the function:

```

myHtml = do br empty
        p
        text "Hello World"

```

will generate the following HTML:

```

<br>
<p>HelloWorld</p>

```

For each HTML tag WASH/HTML defines a correspondent function with the same name as the tag. As we saw in the above example, the function for `
` tag is with name **br**, and for `<p>` is **p**. The text that can appear in the body of a tag is generated with **text**. In case we do not want any content of a tag, then the function **empty** is used.

6.3 HaXml

HaXml is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell. From HaXml, the *Web Cube* prototype uses the separate error-correcting parser for HTML. Unlike the one for XML documents, it must include a certain amount of error-correction to account for HTML features like self-terminating tags, unterminated tags, and incorrect nesting. The HTML parser uses a slightly extended version of the Hutton/Meijer parser combinators. Its input is tokenised by the XML lexer.

The XML (HTML) abstract syntax tree defined in HaXML and the function *htmlParse* are of importance for this thesis. *htmlParse* takes HTML content and returns a **Document** structure:

```
htmlParse :: String → String → Document
```

where the first argument is the file name and the second argument is the HTML content to parse. Detailed information about the XML AST can be found in [31].

6.4 QuickCheck

QuickCheck is a tool for automatic testing of Haskell programs. The provided program specification is in the form of test properties which functions should be satisfied. QuickCheck generates a large number of random cases for the input parameters of these properties and succeeds, if the properties hold, otherwise a counter-example is shown. Although this kind of testing cannot guarantee that the program and specification are consistent, it reduces the risk that they are not. It also provides an easy way to recheck consistency after every change to a module.

6.4.1 Properties

The QuickCheck properties are expressed in Haskell, using predefined combinators in the QuickCheck library. Properties are universally quantified over their parameters which must have monomorphic types and the return type has to be *Bool* or in case combinators are used the return type is *Property*. A typical QuickCheck property would be:

```
test_Reverse :: [Int] → Bool
test_Reverse xs = reverse (reverse xs) == xs
```

To run the test we have to execute the following command:

```
> quickCheck test_Reverse
OK, passed 100 tests.
```

QuickCheck generates 100 random cases for the parameter *xs* of the property and reports back that all of them were successfully executed.

The QuickCheck properties can be also conditional and quantified. The conditional properties are defined by the property combinator `==>`. They hold, if the property after `==>` holds whenever the condition does. All cases which do not satisfy the condition are discarded. For example, if we want to test the reverse function only for cases where the list *xs* is not empty, the property will look like this:

```
test_Reverse :: [Int] → Property
test_Reverse xs = (length xs > 0) ==> (reverse (reverse xs) == xs)
```

The quantified properties are defined with the property combinator *forAll*. Being the first parameter *forAll* expects a custom test data generator for the tested type instead of a standard one. This way it is possible to control the distribution of test data. For example:

```
test_Insert :: Int → Property
test_Insert x = forAll orderedList $ \xs → ordered (insert x xs)
```

6.4.2 The class *Arbitrary*

QuickCheck defines a default test data generators for each type using class *Arbitrary*. Instances for the types `()`, `Bool`, `Int`, `Integer`, `Float`, `Double`, `pairs`, `triples`, `quadruples`, `lists`, and `functions` are provided. To use QuickCheck with other types, an instance of class *Arbitrary* that implements the *arbitrary* method has to be made. In case we want to generate random functions over that type, then the implementation of *coarbitrary* method should also be defined.

6.4.3 Observing data

QuickCheck has three property combinators for test cases observation: *trivial*, *classify* and *collect*. *trivial* counts the cases where the given condition is true and reports the percentage of these cases in the total.

```
test_Reverse :: [Int] → Property
test_Reverse xs = null xs 'trivial' (reverse (reverse xs) == xs)
```

classify helps to separate different test cases and label them. The cases that satisfy the given condition are labelled with it.

```
test_Reverse :: [Int] → Property
test_Reverse xs =
  classify (null xs) "Empty list" $
  classify (length xs > 0) "Non-empty list" $
  (reverse (reverse xs) == xs)
```

```
> quickCheck test_Reverse
OK, passed 100 tests.
82% Non-empty list.
18% Empty list.
```

collect property combinator is used for collecting data values. The parameter of *collect* is evaluated for each test case and its result is reported back. The type of this parameter should have an instance of *Show*.

```
test_Reverse :: [Int] → Property
test_Reverse xs =
  collect (length xs) $ (reverse (reverse xs) == xs)
```

```
> quickCheck test_Reverse
OK, passed 100 tests
20% 0
17% 1
```

```

15% 3
14% 6
13% 8
12% 27
9% 5

```

6.5 Haskell

6.5.1 Haskell records

The Haskell records are just a syntactical sugar of the language which have some significant advantages when working with large data structures. The record syntax assigns labels to the fields of a constructor definition in a data declaration. For example the data type *Rec* defined below can be rewritten as a record with three labelled fields (*f1*, *f2* and *f3*) where the order of the fields is of no importance.

Classic Haskell data type:

```
data Rec = Rec Int String String
```

Haskell record:

```
data Rec = Rec { f1 :: Int, f2 :: String, f3 :: String }
```

The advantages of the use of records instead of classic data types in this prototype are:

1. Easy access to the data fields (by name);
2. Use of the fields without worrying about the order they appear in the record;
3. Easy modification of a record, adding/reordering fields without the need to change other affected code;
4. Producing robust programs, because the code for selectors, update, and construction is generated directly by the compiler;

Unfortunately, there are also some disadvantages. The field names are with global scope and cannot be used in more than one record. This is a serious problem which can be solved only by renaming the field names, thus making them unique. In the *Web Cube* prototype the concrete, test and abstract states (Haskell records) have to be generated based on the same state variables. In this case the prototype generates different field names for each state simply by adding a prefix to the variable.

Here is a small example how record structures are usually used in the prototype. The example shows a declaration of two records, assigning values to the fields, and use of the records.

```

data Rec1 = Rec1 {r1_f1::Int, r1_f2::String}
data Rec2 = Rec2 {r2_f1::Int, r2_f2::String}

use = let r1 = Rec1 { r1_f1 = 1, r1_f2 = "one"}
      r2 = Rec2 { r2_f1 = 2, r2_f2 = "two"}
      in do putStrLn (show (r1_f1 r1) ++ ": " ++ show (r1_f2 r1))
          putStrLn (show (r2_f1 r2) ++ ": " ++ show (r2_f2 r2))

```

Full description of the Haskell record system can be found in [24] and [25].

6.5.2 Monads

To understand some of the work in this thesis, some knowledge about *monads* is required. The monad can be viewed as a state transformer. It receives an input state, makes a computation over this state and returns the result from the computation and the transformed state. Monads are used for sequential computations. They can be combined with the combinators `>>` and `>>=`. These combinators feed the result state of their left hand argument to the input of their right hand argument. Useful monads are `MonadIO`, `MonadState` and `STM Monad`. A tutorial about monads can be found on <http://www.nomaware.com/monads/html/>.

6.5.3 Concurrent Haskell

Concurrent Haskell [27] is an extension of Haskell 98. It provides a new primitive **forkIO** which starts a concurrent process. The synchronization between the threads is done with the help of *transactional variable* of type **TVar a** [26]. Read/write operations of **TVar** can be done with the **readTVar** and **writeTVar** functions. Both take a **TVar** and return a **STM monad**. In order to be able to expose the **STM actions** to the system, a function *atomically* is used:

```
atomically :: STM a → IO ()
```

Atomically takes a memory transaction, of the **STM a** type, and delivers an I/O action. The important here is that the transaction runs atomically with respect to all other memory transactions. In this way it is impossible to read or write accidentally a **TVar** without the protection of *atomically*.

A safe communication between threads can also be done with a global variable called **MVar**. This variable is implemented on top of **TVar**. The only difference is that **MVar** can be either empty or full with a value. The **takeMVar** function reads the value and leaves the **MVar** empty which will block other threads that are trying to read it. A **putMVar** on an empty **MVar** leaves it full. There is also a **readMVar** which will read the

value without blocking the variable.

In the *Web Cube* prototype both **TVar** and **MVar** are used for thread synchronization.

Chapter 7

Testing

Specifying the software before its actual implementation is a good idea. It gives much better ground when we test the implementation later.

Basically, testing compares the actual outputs of a program to the expected ones. A *positive test* aims at showing that a product can meet its specification. Of course, this only validates the product against the specified test cases. Since it is infeasible to test exhaustively all cases, sometimes it is more efficient to do *negative tests*. Such a test aims at 'crashing' a system — that is, at finding a case which violates the specification.

A very popular testing technique is the *black-box testing*. The tester looks at the tested program as a 'black box': only its input, output and specifications are visible. All test cases are derived from the specification. Various inputs are executed during the testing; the outputs are then compared against the specification. In particular, no implementation details (of the tested program) are exploited.

Web Cube supports automated testing. Basically this is done by randomly generating inputs. The test engine performs black box testing where components are tested against their contracts. Negative tests are used for the testing of temporal properties (which a contract may specify) — we will say more about this later.

7.1 Testing of Refinement

Consider two programs called f and $shift$; the first takes the role of an implementation, the second is f 's specification. The term that we will also use here is that $shift$ is called the *abstraction* of f , and f is the *refinement* of $shift$ — the term 'abstraction' is used in general to refer to a mechanism for reducing details in order to concentrate on the essentials, whereas refinement adds more details. The program f could be part of a component, e.g. an action, and $shift$ is its corresponding abstraction in the component's contract.

For simplicity, let us view f and $shift$ as functions from states to states. More precisely, $f :: C \rightarrow C$ and $shift :: A \rightarrow A$. As an abstraction $shift$ typically operates on a more abstract state structure, which implies that there is an injection function $inject$ from C to A . The injection property can be characterized by:

Definition 7.1.1 : INJECTION

$$(\forall a : a \in A : abstr (inject\ a) = a)$$

where $abstr$ is the reverse of $inject$ that takes elements in C to elements in A . The states in A are called *abstract states*, and those in C are called *concrete states*.

We will denote the relation between f and $shift$ with $shift \sqsubseteq f$. There is a number of ways to define this relation — see below. They influence the way f is tested. Another way to look at the definitions is that they specify different scenarios to test f .

Definition 7.1.2 : $\forall f : shift \sqsubseteq^{CC} f$ iff $(f = inject \circ shift \circ abstr)$

Definition 7.1.3 : $\forall f : shift \sqsubseteq^{AC} f$ iff $(f \circ inject = inject \circ shift)$

Definition 7.1.4 : $\forall f : shift \sqsubseteq^{AA} f$ iff $(shift = abstr \circ f \circ inject)$

Definition 7.1.5 : $\forall f : shift \sqsubseteq^{CA} f$ iff $(abstr \circ f = shift \circ abstr)$

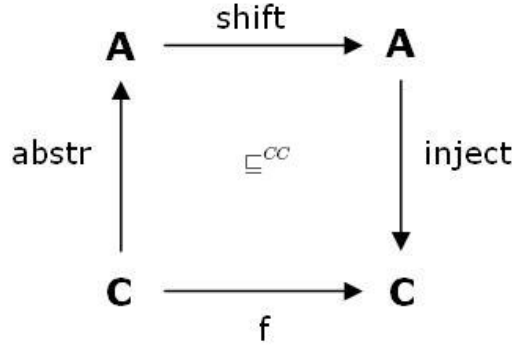
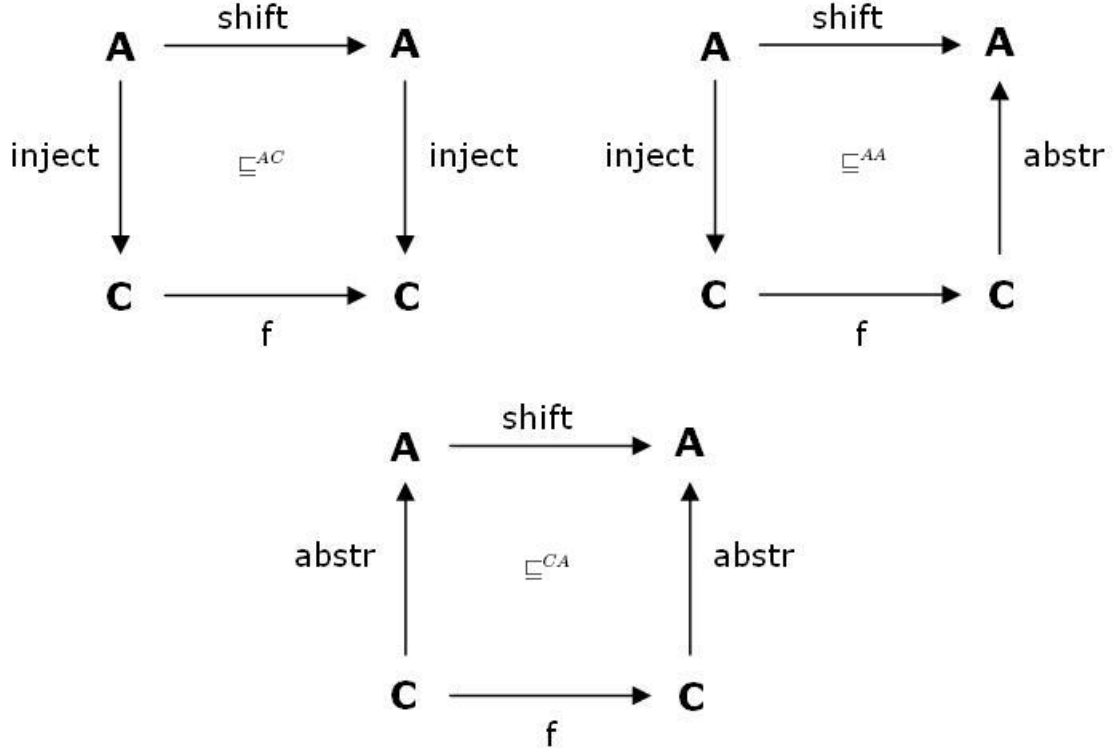


Figure 7.1: Commuting diagram $shift \sqsubseteq^{CC} f$

We can also use a diagram like the one in Figure 7.1 to illustrate the definitions. It is called a *commuting diagram*. The arrows represent functions, and the nodes are types, representing either the domain or the range of the functions. Appending arrows into a path correspond to function compositions, and paths that commute correspond to composite functions which are equivalent. The different definitions above correspond to the different ways the four arrows (f , $inject$, $shift$, and $abstr$) can commute, as shown in the diagrams

Figure 7.2: Commuting diagrams: $shift \sqsubseteq^{AC} f$, $shift \sqsubseteq^{AA} f$, $shift \sqsubseteq^{CA} f$

in Figure 7.2.

We can use automated testing to validate each of the above relations. For example, the \sqsubseteq^{AA} relation can be tested by randomly generated abstract states a (values from A) and then checking if the result of $(abstr \circ f \circ inject) a$ is equal to that of $shift a$. However $inject$ (being a function) only maps a to a single $c \in C$, whereas there may be other states, e.g. c' , in C that corresponds to the same a . So effectively, AA will only test over the subset of C which is the range of $inject$. The rest of C will never be tested. CA is therefore better, as it ranges over all values in C as inputs. Notice that CA implies AA . The downside of CA is that we have to be able to directly generate concrete states, which is sometimes impossible, e.g. if f itself does not offer an interface to do so.

We actually generalize the \sqsubseteq relation a little bit by allowing $shift$ to be a relation rather than function, so that it can be partial and non-deterministic. It also allows the use of Hoare triple to specify $shift$. The definition of the relation \sqsubseteq has to be accordingly adapted. For example, that of AA and CA now become:

Let $relShift :: A \rightarrow A \rightarrow Bool$ be a relation over A .

Definition 7.1.6 : $\forall a : a \in A : \text{relShift} \sqsubseteq^{AA} f$ iff $\text{relShift } a ((\text{abstr} \circ f \circ \text{inject}) a)$

Definition 7.1.7 : $\forall c : c \in C : \text{relShift} \sqsubseteq^{CA} f$ iff $\text{relShift } (\text{abstr } c) ((\text{abstr} \circ f) c)$

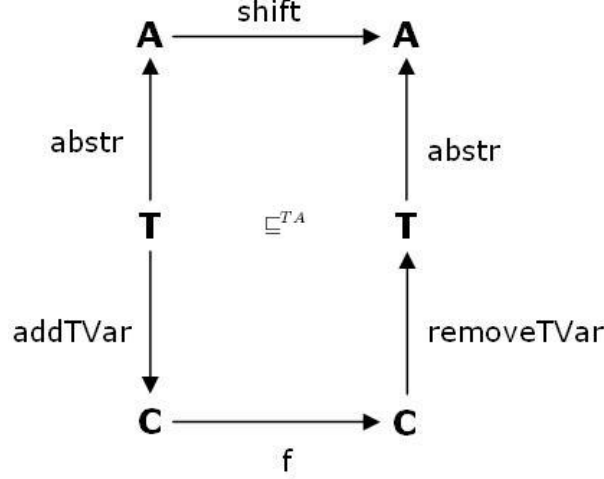


Figure 7.3: *Web Cube* Testing

As we will see in 8.2, the concrete state of a component will be implemented by packing it inside the `TVar` type in Haskell. It is used because it is part of the transactional memory support we use to implement atomic actions. However the direct generation of `TVar` values (which we do via `QuickCheck`) is problematical. What we did was to introduce a new layer of states (called *test states*), which is isomorphic to the state packed inside `TVar`, and then drive the testing from these test states. The diagram in Figure 7.3 shows the scenario. T is the type of test states. The `addTVar` function injects a test state to the corresponding `TVar`. The reverse is `removeTVar`, which extracts the test state from a `TVar`:

```
addTVar :: T -> TVar C
removeTVar :: TVar C -> T
```

In the implementation `addTVar`, and similarly `removeTVar`, would actually be of the type $T \rightarrow IO (TVar C)$ and $IO (TVar C) \rightarrow T$.

Finally the definition of the relation \sqsubseteq takes the form:

Definition 7.1.8 : $\forall t : t \in T : \text{relShift} \sqsubseteq^{TA} f$ iff

$$\text{relShift } (\text{abstr } t) ((\text{abstr} \circ \text{removeTVar} \circ f \circ \text{addTVar}) t)$$

7.2 Testing action

An action of a component is specified in the component's contract by a pair of predicates. For example:

```
action move {
    size(in_) > 0
    assert
    size(in_) == 0
}
```

The predicate to the left of the keyword `assert` specifies the action guard. The predicate to the right specifies the effect of the action, if it is executed. The action skips implicitly, if the guard is false.

Such a specification is then translated to a predicate comparing pre- and post-states:

```
predMove :: AbstState → (AbstState, ()) → Bool
predMove preState (postState, _)
    = if (size (abst_in_ preState)) > 0
        then (size (abst_in_ postState) == 0)
        else (preState == postState)
```

This predicate takes the role of *relShift* in Definition 7.1.8. Since an action cannot have parameters and return value, the predicate does not contain such checks. Notice that the implicit skip in the original specification is represented by the equality between `preState` and `postState`.

7.3 Testing Method

A component's method takes parameters and returns a value, which also has to be taken into account when comparing the method to its specification. This can be easily done by extending the types of *f* and *shift*. For simplicity, assume that *f* has one parameter *v* of type *V* and returns a value of type *R*. Now the definition of TA becomes:

Definition 7.3.1 :

Let $relShift :: V \rightarrow A \rightarrow (A, R) \rightarrow Bool$ be a relation over *T*.

$\forall t : t \in T : relShift \sqsubseteq^{TA} f$ iff

$$relShift \ v \ (abstr \ t) \ (((abstr \circ removeTVar \circ f \circ addTVar) \ t), ret)$$

For example a method is specified in a Web Cube contract:

```

method bool vote(string v)
{
  if (open) then {
    in_ <- insert(v)
  };

  return open
}

```

From such a specification a predicate encoding *shift* (as a relation) will be constructed:

```

predVote :: String → AbstState → (AbstState, Bool) → Bool
predVote v preState (postState, retval)
  = let shift = AbstState { ... expected state ... }
    in (postState == shift) ∧ (retval == (abst_open postState))

```

Note the parameter *v* and *retval*. They show that this predicate also ranges over the parameters and returns value of the method *vote*.

7.4 Invariant

A predicate *i* is a strong invariant of a component *P*, if it holds initially, and is maintained by every action and method in *P*:

Definition 7.4.1 : $P \models \text{sinv } i = P.\text{init} \Rightarrow i \wedge (\forall a : a \in (P.\text{meths} \cup P.\text{acts}) : \{i\} a \{i\})$

The invariant of a component is specified in its contract, though this invariant is thus specified with respect to the abstract states. It is then transformed to a test predicate such as the one below — it corresponds the invariant in the contract of *VoteServer* component shown in Appendix E.

```

predInvariant :: AbstState → AbstState → Bool
predInvariant preState postState
  = (and [((member v (abst_votes preState))) ∧ ((isValid v ))
          | v <- toList (abst_votes preState)])
  ==>
  (and [((member v (abst_votes postState))) ∧ ((isValid v ))
          | v <- toList (abst_votes postState)])

```

This predicate takes the role of *relShift* in Definition 7.1.8. Notice that the invariant is translated twice: one checking the pre-state and one for the post-state. The above predicate is then tested against each action and method in the component.

7.5 Temporal properties

A contract may specify some temporal properties. Only temporal properties expressed in terms of *leadsto* and *unless* operators are supported, because Seuss logic supports only these. In model checking temporal properties can be checked via the so-called Büchi automata. It is known that every LTL temporal property can be described by a Büchi automaton [7]. If P is a reactive program and ϕ is a temporal property, we can check if ϕ is a property of P by checking that there is no execution that is both an execution of P and of N_ϕ , where N_ϕ is the Büchi automaton representing the negation of ϕ . We will borrow this idea for testing. For this purpose an extra library (WebCube.Util.TestTemporal) was developed and included in the *Web Cube* prototype.

A finite automaton (or finite state machine) can be used to accept sentences. Figure 7.4 shows an example; let's call it M . Such an automaton is described by a set of states, transitions, and an alphabet of input symbols. Some of the states are marked as initial states, and some as final states. Such a machine can be thought of as reading a finite sentence as input. The machine starts from an initial state and the input sentence is consumed one symbol at a time. The current symbol can be consumed, if M allows a transition from the current state to another state via the current symbol, else the machine gets stuck.

Implicitly in this example the automaton operates on the alphabet $A = \{p, q\}$. A^* denotes the set of all finite sentences that can be made using the symbols in A . A sentence $s \in A^*$ is accepted by M , if M can completely consume it, and if it ends up then in one of its final states.

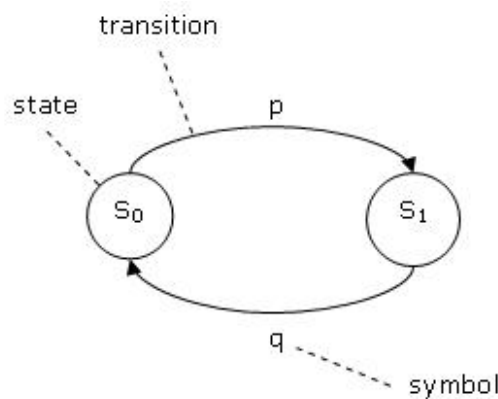


Figure 7.4: Automaton

A Büchi automaton is an extension of a finite state automaton to accept infinite sentences. A Büchi automaton looks the same as a finite automaton (that is, it can be described in the same way), but it has a different accepting criterion. Let B be a Büchi automaton with F as the set of final states. The run of a Büchi automaton is infinite, and

moreover it visits F infinitely very often. An infinite run consumes an infinite sentence. A sentence that can be consumed by a run of B is called accepted by B . Since, as in the finite state automaton, B still has a finite number of states, notice that the existence of a run implies that B must have a cycle.

7.5.1 Testing with omega-Automaton

We can check if a program P satisfies a property ϕ by checking if every execution of P satisfies ϕ . This is called positive checking. Alternatively we can also check if no execution of P satisfies $\neg\phi$, which is called negative check. It looks like it does not matter much which form is used, but it does. When implemented, a positive check would require us to investigate all executions. On the other hand, a negative check only requires us to find a counter example. The latter is what we implement in *Web Cube*.

As already said, a Büchi automaton can be used to describe a temporal property. It is easier to explain this in terms of LTL formulas. The Seuss property $p \mapsto q$ corresponds to the LTL formula $\Box(p \rightarrow \Diamond q)$. Its negation $\neg\Box(p \rightarrow \Diamond q)$ can be rewritten to $\Diamond(p \wedge (\Box\neg q))$. The (negative) formula is described by the Büchi automaton in Figure 7.5; S_0 is an initial state and S_1 is an accepting state. The input symbols are now state predicates like p and q .

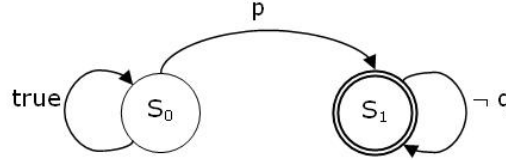


Figure 7.5: Non-deterministic Büchi automaton: $\neg(p \mapsto q)$

If P is the program to be checked, the idea now is to run P in *lock step* with the Büchi automaton in Figure 7.5 (let us call it B). It means that we run both P and B . Both begin in their initial states. Suppose (σ, s) are the current states of P respectively B . A lock-step run makes a transition to a new state (σ', s') only if the transition from σ to σ' is allowed by P , and there is a transition from s to s' in the Büchi automaton B labelled by a state predicate P , such that P holds in s' . Notice that a lock-step run is always a run of P itself. Now, if this run visits the accepting state of B infinitely often, then this run also satisfies the temporal property specified by B . In the example of Figure 7.5 such a run would be the counter example of the property $p \mapsto q$. This is nice, but the problem is still that this run is an infinite run — we cannot implement such a run. However if P and B have finite states then such a run can only exist if it is cyclic. So, it is sufficient to find a run that cycles over the accepting states of B . This can be implemented. For the purpose of detecting a cycle, we maintain a history of states (h) ,

which has been passed during a run. Let s is the current state; a cycle is found when $s \in h$.

In the case when P is a component, so it is expected to behave as a Seuss box, its runs must also be *fair*. That is, a cycle that contains all actions of the tested program and it is a subset of the execution history. Consequently, when we find a cycle, we also have to check if this cycle is a fair cycle. That is, a cycle in which all actions of P (*actions*) have participated. In order to check this, the history is extended by an action identifier.

Notice that the Büchi automaton in Fig. 7.5 is non-deterministic. This is problematical for the implementation of the lock step execution, because it may force us to backtrack the execution. This leads us to the thought of converting the Büchi automaton to a deterministic one.

In the case of finite automaton, we could convert each non-deterministic automaton into a language-equivalent deterministic one. Unfortunately, this is not the case with Büchi automata. The non-deterministic Büchi automata are strictly more expressive than the deterministic ones. Fortunately, such a conversion is still possible for \mapsto and *unless* properties. The automaton in Fig. 7.5 is converted to the one in Fig. 7.6. Furthermore we alter the accepting criterion. A run of the automaton in Fig. 7.6 is: (1) infinite, and (2) it has a suffix where it remains forever in the accepting states. Acceptance of a sentence is defined analogously.

Because of the altered accepting criterion, this automaton is no longer a Büchi automaton. It is still an ω -automaton, which is the term used to refer to the broad class of automata for accepting infinite sentences. It can be shown that the automaton in Fig. 7.6 accepts the same language as that in Fig. 7.5.

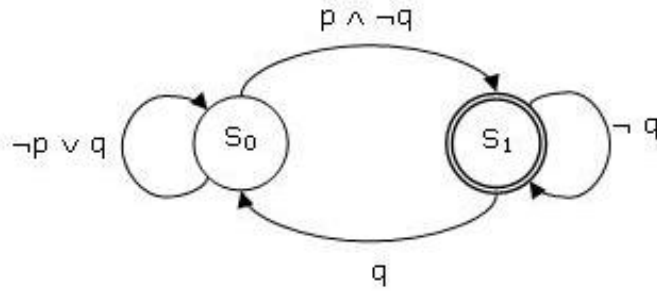
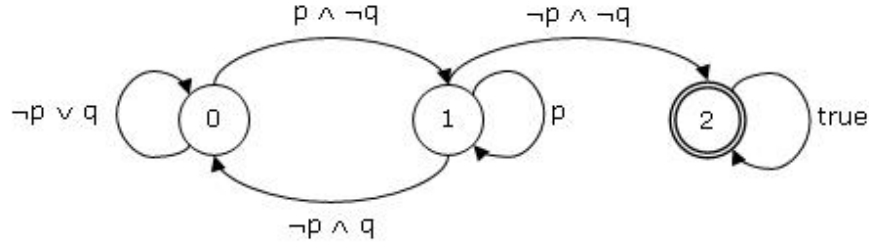


Figure 7.6: Deterministic Büchi automaton: $\neg(p \mapsto q)$

The automaton that corresponds to the negation of p unless q is shown in Fig. 7.7. This automaton uses the same accepting criterion as the one in Fig. 7.6, and as before it has to be executed in lock step with the target program P .

The above approach of checking temporal properties is implemented in the module

Figure 7.7: Deterministic Büchi automaton: $\neg(p \text{ unless } q)$

WebCube.Util.TestTemporal.hs.

Note that what we are doing is testing, not model checking. The essential difference is that we would only test against a relatively small number of runs, whereas in model checking we would in principle explore all runs. As such, testing can find a counter example, but it is not guaranteed it will find one, even if the given property is invalid.

WebCube.Util.TestTemporal.hs implements ω -automatons as a function of type:

```
type BM state = BMState state -> state -> IO()
```

where *BMState* is a shared global variable:

```
type BMState state = IORef (Int, [state])
```

The first element in the tuple is an identifier of an automaton state, and the second element is a history of the program states during execution.

We define two automaton: *neverLTO* and *neverUNL*. The first implements the automaton in Fig. 7.6 for checking $\neg(p \mapsto q)$, and the second implements the automaton in Fig. 7.7 for checking $\neg(p \text{ unless } q)$.

A temporal property like $p \mapsto q$ can be represented by a function that takes two predicates, the *p* and the *q*, and returns an ω -automaton:

```
type Pred a = a -> Bool
type TemporalProp state = Pred state -> Pred state -> BM state
```

Since the automaton *neverLTO* and *neverUNL* are already defined, we can use them to express two more automaton namely *neverEV* and *neverSTAB*.

```
neverEV tr q = neverLTO tr (\c -> True) q
neverSTAB tr p = neverUNL tr p (\c -> False)
```


Lock step execution is implemented by the function `lockstep` below. It takes an ω -automaton `machine` as an argument and it is subsequently applied to an action `a`. It can be thought as a wrapping around `a`. It uses a synchronization variable to force the execution of `a` to happen in step locking with a step from `machine`.

```
lockstep sync machine (id, a) =
  do { takeMVar sync
      ; a
      ; machine id
      ; putMVar sync ()
    }
```

To force a lock step execution between a component P and a machine M , we map `lockstep M` on all actions of P and then simply run each 'wrapped' action repeatedly in a separate thread.

Because the actions are run in parallel in different threads, once running we have to wait until all of them complete their execution. Since a test fails if just one of the threads fails, this behaviour can be optimized. A new data type *TestStatus* is introduced in order to be able to handle the results of the different threads. It is also global like the *BMState*, and it is implemented based on *MVar*. The first element in the tuple of *TestStatus* is a key/value pair list that collects threadId and isReady flag. The second element in the tuple is a boolean storing if a test from all threads failed (called result flag). If just one test fails, then the rest of the threads exit prematurely their execution.

```
type TestStatus state = MVar ((Map ThreadId Bool), Bool)
```

To access the *TestStatus* there are a number of helper functions available such us:

- *initTestStatus* - returns the initial state of the type *TestStatus*. The key/value pair list is set to empty and the result flag initially is set to True;
- *register* - inserts a threadId into the list with isReady flag set to False;
- *failTest* - sets the result flag of testing to False;
- *readyTest* - takes the current threadId and sets the isReady flag for it in the list to True.
- *getTestStatus* - this is the main function which processes the information found in *TestStatus* and waits until all threads are ready. The function waits not only until all threads in the list have isReady flag set to True, but also waits until the length of the list is equal to the number of actions to test.

To demonstrate the use of this library, a small example is given below. It shows a reactive component with two actions and a counter forming the component's state. The first action increments the counter with 2 and the second action resets the counter.

```

action0 state = atomically (
    do { x <- readTVar state
        ; writeTVar state (x+2)
        }
    )

```

```

action1 state = atomically (
    do { writeTVar state 0 }

```

To test the example we give below three simple temporal properties: *prop1* claims that eventually the counter will be equal to 10; *prop2* claims that the counter will remain bigger than 0; and *prop3* claims that the counter's value will remain even. We expect that the tests of *prop1* and *prop2* fail and *prop3* succeed.

```

prop1 tr = neverEV    tr (\(_,s) -> s == 10)
prop2 tr = neverSTAB tr (\(_,s) -> s > 0)
prop3 tr = neverSTAB tr (\(_,s) -> even s)

```



```

C:\Program Files\Visual Haskell\bin\ghci.exe
*TestTemporal> test1
<"initial state",0>
<"a1",2>
<"a2",0>
False
*TestTemporal> test2
<"a1",2>
<"a2",0>
False
*TestTemporal> test3
True
*TestTemporal> _

```

Figure 7.8: Results from running *prop1*, *prop2* and *prop3*

Chapter 8

Generator

In Chapter 5 we have shown an extended example of a *Web Cube* application. We have also shown part of the syntax of *WebCube*. This chapter explains the Haskell codes generated by the *Web Cube* prototype based on the `WebVote` source code shown in Chapter 5.

8.1 Generating WebFunctions

The *WebFunctions* generator can be found in modules `WebCube.Generator.WebFunctions` and `WebCube.Generator.ProcessHTML`. It generates a Haskell implementation of a *Web Cube* application. More precisely, the implementation is done via the *WebFunctions* library by translating the *Web Cube* application to a *WebFunctions* application. The first part of the generated Haskell file specifies the necessary imports. For `WebVote` these are:

```
module Main where

import WASHHTML.HTMLMonad98
import WebFunctions.WebFunctions
import WebFunctions.Components.WFAccessors
import WebFunctions.Components.HTML.WFFormComps
import Prelude hiding (head)

import Data.Set hiding (empty)
import Control.Concurrent(forkIO)
import Control.Monad.State(liftIO)

import VoteServer
```

The last line imports `VoteServer` which is used in `WebVote` as a *Web Cube* resource.

Subsequently the `main` function is generated followed by *WebFunctions* configurations which are needed to start web applications. Since resources are shared between *Web*

Cube applications, the `main` function starts by creating all resources (and thus their states too) and running their actions. Then the applications are started. For each Web Cube application, its corresponding *WebFunctions* application is generated. In *WebFunctions* an application is started by calling the function `wfStart`. The components that constitute the application are specified in its configuration. Among other things, it specifies a default action, which is the starting point of the application. For a *Web Cube* application, the starting function is the `home` method of the cube `home`. In our example we will only start one application, so `main` looks like:

```
main = do putStrLn "***** Generated by Web Cube *****"
        voteServerState_ <- createVoteServerState
        runVoteServer voteServerState_
        wfStart (myConfig voteServerState_ ) 'catch' print

myConfig voteServerState_
  = defaultConfig (defAction voteServerState_ ) []
defAction voteServerState_
  = mkDefaultAction (do webVote_ <- createWebVoteState voteServerState_
                      return $ mHomeHome webVote_)
```

Each cube in an application may have a state. Furthermore it is also necessary for each method in the cube to maintain a state, which is a trick we use to allow parameters to be passed to the method implementation in *WebFunctions*. Methods are implemented as *WebFunctions* actions, which unfortunately are parameterless, though they can read from states. The state is represented by a Haskell record. An example is shown below, which belongs to the cube `home` of `WebVote`. Furthermore, a function `create<state name>` is also created which is used to initialize a new instance of that state.

```
data HomeState = HomeState { m_home :: HomeHomeState
                             , m_vote :: HomeVoteState
                             , m_stop :: HomeStopState
                             , m_info :: HomeInfoState}

createHomeState
  = do m_home_ <- createHomeHomeState
      m_vote_ <- createHomeVoteState
      m_stop_ <- createHomeStopState
      m_info_ <- createHomeInfoState
      return $ HomeState { m_home = m_home_
                          , m_vote = m_vote_
                          , m_stop = m_stop_
                          , m_info = m_info_ }
```

As already mentioned, the HTML from the *Web Cube* respond is parsed with the help of *HaXml* library. The result abstract syntax tree is used to generate the correspondent *WebFunctions* code. *WebFunctions* uses the WASHHTML library to generate HTML, so the *Web Cube* generated code is a combination of WASHHTML and *WebFunctions*. For example, the string in the content of a tag, as well as the `br` and `p` tag are directly translated to the WASHHTML `text`, `br` and `p`. From the other hand local calls are translated to a `wfcAction`; HTML forms to `wfcForm` and input submit tags to `wfcSubmit` all of which are *WebFunctions* components. The code below is the generated Home method of the WebVote application shown in Chapter 5:

```
mHomeHome webVoteState = pageTemplate $ do
  wfcForm
    (do text "Enter your vote: "
        v_ <- wfRead $ (v (m_home (c_home webVoteState)))
        wfcFormComponent
          (λattval → wfWrite (v (m_home (c_home webVoteState)))
                        attval)
          (attr "value" v_ ## attr "type" "text"))

    wfcSubmit
      (attr "VALUE" "Vote")
      (mkSessionAction
        (do v_ <- wfRead $ (v (m_home (c_home webVoteState)))
            return $ mHomeVote v_ webVoteState)))
      (mkDirectAction (return $ mHomeHome webVoteState) "HomeHome")

  p (do wfcAction
        (text "Vote info")
        (mkSessionAction (return $ mHomeInfo webVoteState)))
  p (do wfcAction
        (text "Stop voting")
        (mkSessionAction (return $ mHomeStop webVoteState)))
```

Unfortunately, our *Web Cube* prototype does not implement the full range of HTML tags. Partly because the underlying *WebFunctions* support is also incomplete. In principle, more HTML support can be easily added.

8.2 Generating component skeleton

Sometimes the developer of a *Web Cube* application is also the developer of the components used in the application. To help the developer, the *Web Cube* prototype can also generate a skeleton of a component, based on a given contract. This encourages the developer to

write first a contract before implementing the component.

The skeleton generator can be found in the module *WebCube.Generator.Component*. Consider again the component *VoteServer*. We assume the contract shown in Appendix E. Let's take a look at the generated skeleton.

First the component state is generated. As mentioned earlier, all the component's variables that form the state are wrapped with *TVar*. In this way the state can be shared between different concurrent actions. A function *create<state name>* is also generated to initialize the state.

```
data VoteServerState = VoteServerState
  { in_    :: TVar (Set String)
  , votes :: TVar (Set String)
  , open  :: TVar (Bool)
  }
deriving Eq

createVoteServerState :: IO VoteServerState
createVoteServerState
  = do atomically (do in__  <- newTVar Data.Set.empty
                    votes_ <- newTVar Data.Set.empty
                    open_  <- newTVar False

                    return $ VoteServerState { in_    = in__
                                              , votes = votes_
                                              , open  = open_
                                              }

                    )
```

A compulsory part of a component is the function *run<component name>*, which will fork a new thread for each action implemented in the component.

```
runVoteServer :: VoteServerState → IO()
runVoteServer state = do forkIO (actionMove state)
                    forkIO (actionValidate state)
                    return()
```

For each action an extra wrapper is generated which will run the action repeatedly (and forever), as required by Seuss' execution model. For example:

```
actionMove :: VoteServerState → IO()
actionMove state =
  do forever (move state)
```

where `forever` is

```
forever :: Monad m => m a -> m a
forever act = do { act; forever act; }
```

The component can make calls to Haskell functions or to functions provided by the component itself. The names of all these function calls are collected in a comment in the generated skeleton as a reminder to the programmer to import libraries or to implement functions.

To support (automated) testing, the skeleton also makes a testing section which specifies a type representing the so-called test states, an instance of `Arbitrary`, and an implementation of `addTVar` and `removeTVar`. As already mentioned, a component state is implemented as values stored inside `TVar`. For the purpose of testing, we need a copy of such a state, but without the `TVar` wrapping; this is called *test state*.

```
data TestState = TestState { test_in _ :: Set String
                           , test_votes :: Set String
                           , test_open  :: Bool
                           }
  deriving (Show, Eq)
```

Instances of `Arbitrary` are needed by `QuickCheck` for randomly generating test data. Three instances are included in the skeleton: that of `Char`, that of `Set`, and that of the type representing test states. For the skeleton of our `VoteServer` component they are:

```
instance (Ord a, Arbitrary a) => Arbitrary (Set a) where
  arbitrary = liftM fromList arbitrary
  coarbitrary sl = coarbitrary (toList sl)

instance Arbitrary TestState where
  arbitrary = liftM3 createTestState arbitrary arbitrary arbitrary
```

Quite often the implementation of a component will have a more detailed state structure than the shown in its contract. In order to implement this, the developer can simply extend the data type generated in the template for representing the component's states (the data type `AbstState` in the example above). He also has to adjust the implementation of `create<component state>`, `TestState`, `createTestState`, the instance of `Arbitrary` over `TestState`, `addTVar` and `removeTVar` which are all affected by the change of the state structure.

8.3 Generating tests

The *Web Cube* prototype also provides a test generator. It can be found in the module `WebCube.Generator.Testing`. When given a contract, it generates a test file named `<contract name>Test.hs`. When given a component which implements the contract, this file can test if the component respects the contract. The test is fully automatic, and it is based on random generation of inputs and states. The generated test file contains 3 sections. A representation of the component's states is generated first. Since we only use the information in the contract, these states are the component's abstract states. Next, test codes against invariant, method specifications, and smodel are generated. Finally test codes against temporal properties in the contract are generated.

```
data AbstState = AbstState { abst_in_    :: Set String
                             , abst_votes :: Set String
                             , abst_open  :: Bool
                             }
deriving (Show, Eq)
```

Recall Figure 7.3; it shows *Web Cube* testing scheme. There are three sorts of states involved. These sorts of states are: concrete, test, and abstract states. The test generator does not provide the definition of test states, so the programmer has to define it himself. However, if the component is written from a skeleton generated from our skeleton generator, it will already include a definition of these test states.

The testing scheme in Figure 7.3 also requires five functions: *addTVar*, *removeTVar*, *abstr*, *shift* and *f*. The first two are also included in a generated skeleton; *abstr* is generated as part of the test file; *f* is the actual tested function from the implementation and *shift* is a generated abstraction of *f*.

```
abst :: TestState → AbstState
abst testState = AbstState { abst_in_    = test_in_    testState
                             , abst_votes = test_votes testState
                             , abst_open  = test_open  testState
                             }
```

For each method and action specification in a *Web Cube* contract, as well as the invariant, a QuickCheck predicate is generated. This property can be run against actual methods and actions of the component and will cause QuickCheck to perform automated test against the property. For example the QuickCheck predicate representing the specification of the *vote* method is the following:

```
predVote :: String → AbstState → (AbstState, Bool) → Property
predVote v preState (postState, retval)
```



```

= let shift = AbstState { abst_in_ =
                        if (abst_open preState)
                          then insert v (abst_in_ preState)
                          else (abst_in_ preState)
                        , abst_votes = (abst_votes preState)
                        , abst_open = (abst_open preState)
                      }
  in label "smodel vote" ((postState == shift)
                        ^ (retval == (abst_open postState)))

```

Notice that the predicate takes two abstract states as arguments. The first one, `preState`, is used to pass the initial abstract state of the tested function. The other, `postState`, is used to pass the final abstract state of the tested function when it is executed on the initial state and passed `v` as the argument. Additionally, we also pass the value returned by the tested function to `predVote`. This information is paired with `postState`. The predicate `preVote` compares all these values. It checks in particular whether `postState` and `retVal` satisfy the expected values, e.g. the value `shift` in the code represents the expected `postState`.

Run functions are also generated to run concrete methods (the methods of the actual component) on a given initial test state. A run function will return the final test state and the value returned by the object method.

```

run1 func a testState = unsafePerformIO
  (do cs <- addTVar testState
    ret <- func a cs
    ts <- removeTVar cs
    return (ts, ret))

```

This run functions are used in a test wrapper, which provides the initial state and the result state to a predicate.

```

testSMod1 func pred a testState =
  let (state, retval) = run1 func a testState
  in pred a (abst testState) ((abst state), retval)

```

Finally, `smodel` is a collection of quickCheck runs of the methods and actions specified in the contract with the correspondent predicates. One predicate is generated for the invariant, and it is checked against all methods and actions.

```

smodel = do ...
        ...
        quickCheck $ testSMod1 (VoteServer.vote) (predVote)

```

The technique described in Section 7.5.1 is used to test Seuss temporal properties, in particular progress properties. An omega-automaton is generated from each *leadsto* property in the contract. For example, the contract of `WebServer` specifies this progress property:

```
propProgress1 res v = neverLT01 res v
  (\v state -> ((&&) ((&&) (member v (abst_in_ (abst state)))
                    (isValid v ))
                (abst_open (abst state))))
  (\v state -> (member v (abst_votes (abst state))))
```

Notice that this is not a plain *leadsto* property, but a universally quantified *leadsto* property. The code calls the function `neverLT01` which implements the Büshi automaton that corresponds to the negation of a universally quantified *leadsto* property.

A `runProgress` function is used to run the Büshi automaton. This is defined in the `TestTemporal` library. This function expects a Büshi automaton as an argument, and a list of reactive actions and methods against which the automaton is to be checked. `QuickCheck` is used to randomly generate the arguments of the tested methods during the run.

Chapter 9

Final remarks

9.1 Future work

In this thesis we implement a *Web Cube* prototype. It is its first prototype. Some issues are still left open for future work. The current prototype supports validation via automated testing. Verification would provide a stronger result, but this is not supported. Generally, one can extend the generators to produce, e.g. HOL fragments for verification in the theorem prover. The prototype is also incomplete in its support for generating HTML components. Right now only HTML components needed for the demo examples are implemented. Extending the support is not difficult in general. The prototype was not tested for performance, too.

9.2 Conclusion

The implementation of the *Web Cube* prototype tries to reuse as much as possible of the existing software. The prototype is implemented in Haskell and it includes a *Web Cube* parser, a syntax validator, and a set of generators. The tool can generate not only *WebFunctions* implementation (of a *Web Cube* application), but also a component's skeleton and a code to perform automated testing. Since components are implemented in Haskell, they are easy to integrate with the generated *WebFunctions* implementation. The use of *WebFunctions* is mostly useful for hiding the session management from the programmer's concern. Resources can be shared over multiple sessions.

Appendix A

Web Cube Grammar

$\langle name \rangle ::= \text{String}$

$\langle qualified\ identifier \rangle ::= \langle name \rangle (\text{'.'} \langle name \rangle)^*$

$\langle webcube \rangle ::= \langle application \rangle^* \langle contract \rangle^*$

$\langle import \rangle ::= \text{import } \langle qualified\ identifier \rangle \text{';'}$

$\langle application \rangle ::= \langle import \rangle^* \text{application } \langle qualified\ identifier \rangle \text{'{' } \langle resource \rangle^* \langle cube \rangle^* \text{'}'}$

$\langle resource \rangle ::= \text{resource } \langle name \rangle \text{'=' } \langle qualified\ identifier \rangle \text{';'}$

$\langle variable \rangle ::= \langle type \rangle \langle name \rangle \text{'=' } \langle expression \rangle \text{';'}$

$\langle type \rangle ::= \text{void} \mid \text{bool} \mid \text{int} \mid \text{string} \mid \text{intset} \mid \text{stringset} \mid \text{boolset}$

$\langle cube \rangle ::= \text{cube } \langle name \rangle \text{'{' } \langle variable \rangle^* \langle method \rangle^* \text{'}'}$

$\langle method \rangle ::= \text{method } \langle type \rangle \langle name \rangle \text{'(' } \langle parameter \rangle^* \text{'')' } \langle statement \rangle$

$\langle parameter \rangle ::= \langle type \rangle \langle name \rangle$

$\langle contract \rangle ::= \text{contract } \langle name \rangle \text{'{' } \langle smodel \rangle \langle invariant \rangle \langle progress \rangle^+ \text{'}'}$

$\langle smodel \rangle ::= \text{smodel } \langle box \rangle$

$\langle box \rangle ::= \langle name \rangle \text{'{' } \langle variable \rangle^* \langle method \rangle^* \langle boxaction \rangle^* \text{'}'}$

$\langle boxaction \rangle ::= \text{action } \langle name \rangle \text{'{' } \langle expression \rangle \text{assert } \langle expression \rangle \text{'}'}$

$\langle invariant \rangle ::= \mathbf{inv} \text{ '}' \{ \langle expression \rangle \} \text{ '}'$
 $\langle progress \rangle ::= \mathbf{progress} \text{ '}' \{ \langle expression \rangle \} \mathbf{leadsto} \langle expression \rangle \text{ '}'$
 $\langle statement \rangle ::= \langle skip \rangle \mid \langle return \rangle \mid \langle respond \rangle \mid \langle declaration \rangle$
 $\quad \mid \langle sequence \rangle \mid \langle if \rangle \mid \langle assignment \rangle$
 $\langle skip \rangle ::= \text{ '}' \{ \} \text{ '}'$
 $\langle return \rangle ::= \mathbf{return} \langle expression \rangle$
 $\langle respond \rangle ::= \mathbf{respond} \text{ '(' string ')'}$
 $\langle declaration \rangle ::= \text{ '}' \{ \langle type \rangle \langle name \rangle \text{ '=' } \langle expression \rangle \text{ ';' } \langle statement list \rangle \} \text{ '}'$
 $\langle sequence \rangle ::= \text{ '}' \{ \langle statement list \rangle \} \text{ '}'$
 $\langle if \rangle ::= \mathbf{if} \langle expression \rangle \mathbf{then} \langle statement \rangle \mathbf{else} \langle statement \rangle$
 $\langle assignment \rangle ::= \langle name \rangle \text{ " < - " } \langle expression \rangle$
 $\langle statement list \rangle ::= \langle statement \rangle \text{ (';')?}$
 $\quad \mid \langle statement \rangle \text{ '}' \langle statement list \rangle$
 $\langle expression \rangle ::= \langle simple expression \rangle \langle binary operator \rangle \langle expression \rangle$
 $\quad \mid \langle simple expression \rangle$
 $\langle binary operator \rangle ::= \text{ ' + ' } \mid \text{ ' - ' } \mid \text{ ' / ' } \mid \text{ ' * ' } \mid \text{ ' \& ' } \mid \text{ ' | ' } \mid \text{ " == " }$
 $\quad \mid \text{ ' < ' } \mid \text{ ' > ' } \mid \text{ " < = " } \mid \text{ " = > " } \mid \text{ " ! = " } \mid \text{ " == > " }$
 $\quad \mid \text{ " in " } \mid \text{ " union " }$
 $\langle simple expression \rangle ::= \langle constant \rangle$
 $\quad \mid \langle variable \rangle$
 $\quad \mid \langle unary operator \rangle$
 $\quad \mid \langle call \rangle$
 $\quad \mid \langle old value \rangle$
 $\quad \mid \langle binding \rangle$
 $\quad \mid \langle set \rangle$
 $\quad \mid \langle set comprehension \rangle$
 $\quad \mid \langle parenthesized expression \rangle$
 $\langle constant \rangle ::= \text{Int, String, Float, "true", "false"}$

$\langle \text{variable} \rangle ::= \langle \text{qualified identifier} \rangle$

$\langle \text{unary operator} \rangle ::= '!' \langle \text{expression} \rangle$

$\langle \text{call} \rangle ::= \langle \text{qualified identifier} \rangle '(' (\langle \text{expression} \rangle (',' \langle \text{expression} \rangle)^*)? ')'$

$\langle \text{old value} \rangle ::= \mathbf{old} \langle \text{variable} \rangle$

$\langle \text{binding} \rangle ::= \langle \text{binder} \rangle \langle \text{parameter} \rangle ":" \langle \text{expression} \rangle ":" \langle \text{expression} \rangle$

$\langle \text{binder} \rangle ::= \mathbf{forall} \mid \mathbf{exist}$

$\langle \text{set} \rangle ::= "\backslash [\langle \text{expression} \rangle^* \backslash]"$

$\langle \text{set comprehension} \rangle ::= "\backslash [\langle \text{expression} \rangle '|' \langle \text{parameter} \rangle "<- " \langle \text{expression} \rangle$
 $\quad ', ' \langle \text{expression} \rangle \backslash]"$

$\langle \text{parenthesized expression} \rangle ::= '(' \langle \text{expression} \rangle ')'$

Appendix B

Web Cube Keywords

<i>application</i>	<i>contract</i>	<i>component</i>	<i>cube</i>
<i>method</i>	<i>action</i>	<i>respond</i>	<i>return</i>
<i>import</i>	<i>resource</i>	<i>smodel</i>	<i>inv</i>
<i>progress</i>	<i>if</i>	<i>then</i>	<i>else</i>
<i>skip</i>	<i>leadsto</i>	<i>forall</i>	<i>exist</i>
<i>old</i>	<i>assert</i>	<i>in</i>	<i>union</i>

Appendix C

Web Cube Prototype: Options

short option	long option	description
-i	-input	input directory
-o	-output	output directory
-h, -?	-help	shows this help information
-v	-validate	validate WebCube
-s	-skeleton	generate component skeleton
-t	-tests	generate tests
-c	-compile	compile generated files
-w	-webfunc	generate WebFunctions

Appendix D

WebVote.cube

```
import VoteServer;

application WebVote {
  resource r = VoteServer;

  cube home
  {
    method void home()
    {
      respond("<form method=post action='home.vote'>
        Enter your vote: <input type='text' name='v'>
        <input type=submit value='Vote'>
        </form>");
      respond("<p><a href='home.info'>Vote info</a></p>");
      respond("<p><a href='home.stop'>Stop voting</a></p>")
    }

    method void vote(string v)
    {
      if (r.vote(v)) then
        respond("<p>Your vote <b><expr> v </expr></b> was processed</p>")
      else
        respond("<p>Voting is closed.</p>");

      respond("<p align=center><a href='home.home'>Back</a></p>")
    }

    method void stop()
    {
      dummy <- r.stop();
    }
  }
}
```

```
    respond("<p>The voting is closed.</p>");
    respond("<p align=center><a href='home.home'>Back</a></p>")
}

method void info()
{
    respond("<p>Total votes = <expr> r.info() </expr></p>");

    if (r.isOpen()) then
        respond("<p>Voting is open.</p>")
    else
        respond("<p>Voting is closed.</p>");

    respond("<p align=center><a href='home.home'>Back</a></p>")
}
}
```

Appendix E

VoteServer.ctr

```
contract VoteServer {  
  smodel {  
    stringset in_ = \[\];  
    stringset votes = \[\];  
  
    bool open = true;  
  
    method bool vote(string v)  
    {  
      if (open) then {  
        in_ <- insert(v)  
      };  
  
      return open  
    }  
  
    method int info()  
    {  
      return size(votes)  
    }  
  
    method bool isOpen()  
    {  
      return open  
    }  
  
    method void stop()  
    {  
      open <- false  
    }  
  }  
}
```

```

    action move {
        size(in_) > 0
        assert
            size(in_) == 0
    }

    action validate {
        true
        assert
            isSubsetOf((old votes), votes)
    }
}

inv {
    forall string v : (v in votes) : isValid(v)
}

progress {
    forall string v : skip
        : ((v in in_) & isValid(v) & open)
        leadsto (v in votes)
}
}

```

Appendix F

Definition of terms

Web Cube prototype.(or simply *the prototype*) The tool developed in this master thesis. The prototype parses and validates *Web Cube* sources, generates a component skeleton, test properties and a runnable *WebFunctions* application.

Web Cube application. A server side program that interacts with a client via an HTTP connection. Consists of cubes and resources.

Resource. A state-persistent reactive system or another application. Client cannot interact directly with the resources.

Cube. Defines the set of functionalities over the resources that are available to the client.

Method. A mechanism for the environment to alter the application state. Methods can have parameters.

Action. An action is an atomic, terminating, and non-deterministic state transition. Used to update the resource state under specified conditions. Actions cannot have parameters.

Component. A constituent element of a *Web Cube* application. The *Web Cube* paradigm use a component-based approach. Each *Web Cube* application can be a component of a bigger system/application.

Contract. A component specification written as a black box. Specifies how the component behaves as a reactive system and contains three major sections: an smodel, an invariant and a progress.

State. A snapshot of *Web Cube* application during execution. A state can contain other states and application variables.

Session. The time spent by a single user at *Web Cube* application. The session can con-

tain user specific choices stored in a state.

Programmer. The implementer of the *Web Cube* sources and contracts. The programmer is also a user of the *Web Cube* prototype and generated component skeleton and test properties.

User. Everyone who uses running *Web Cube* application throw a browser.

Terms from *IEEE Standard Glossary of Software Engineering Terminology*:

Test. An activity in which a system or a component is executed under specified conditions, the results are observed or recorded, and some aspect of the system or component is evaluated.

Test case. A set of test inputs, execution conditions and expected results developed for a particular objective, such as to exercise a particular program path or to verify a compliance with a specific requirement.

Verification. (1) The process of evaluating a system or a component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness.

Validation. The process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements.

Proof of correctness. A formal technique used to prove mathematically that a computer program satisfies its specified requirements.

Invariant. An assertion that should always be true for a specified segment or at a specified point of a computer program.

Bibliography

- [1] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, B. Widjaja. *Web Cube: a Programming Model for Reliable Web Applications*. Institute of information and computing sciences, Utrecht university technical report UU-CS-2005-002.
- [2] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra *Web Cube* Dept. of Inf. and Comp. Sciences, Utrecht University, the Netherlands, Instituto Tecnológico de Informatica, Valencia, Spain
- [3] K.M. Chandy and J. Misra, *Parallel Program Design*. Austin, Texas, May 1989
- [4] I.S.W.B. Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. *!UNITY: A HOL theory of general UNITY*. In D. Basin and B. Wolff, editors, Emerging Trends Proceedings of 16th International Conference, Theorem Proving in Higher Order Logics (TPHOL), pages 159-176, 2003. Also available as tech. report No. 187 of Inst. für Inf., Albert-Ludwig-Univ. Freiburg. Available on-line at <http://www.informatik.uni-freiburg.de/tr>.
- [5] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, B. Widjaja. *A Theory for Composing Distributed Components, Based on Mutual Exclusion*.
- [6] I.S.W.B. Prasetya, T.E.J. Vos, A. Azurat, S.D. Swierstra. A UNITY-based Framework towards Component Based Systems
- [7] Gerard J. Holzmann. The Model Checker SPIN. IEEE Transactions on software engineering, Vol 23, No. 5, May 1997
- [8] Robert van Herk. *WebFunctions*. Master thesis, September 30, 2005
- [9] J. Misra. *A Discipline of Multiprogramming*. Department of Computer Sciences, The University of Texas at Austin, March 1999.
- [10] Ingolf Heiko Kruger, *Thesis project. An experiment in compiler design for a concurrent object-based programming language*. Department of Computer Sciences, The University of Texas at Austin, 1996.
- [11] Rajeev Joshi. *Seuss for Java. Language reference*. Department of Computer Sciences, The University of Texas at Austin, February 1998.

- [12] I.S.W.B. Prasetya, A. Azurat, T.E.J. Vos, A. van Leeuwen, H. Suhartanto, *Theorem Prover Supported Logics for Small Imperative Languages*. Technical report UU-CS-2005-046
- [13] *The HOL System. Reference*. For HOL Kananaskis-2, March 8, 2004. Available on-line at <http://www.cl.cam.ac.uk/Research/HVG/HOL/documentation/reference.dvi.gz>.
- [14] *The HOL System. Description*. For HOL Kananaskis-2, March 8, 2004. Available on-line at <http://www.cl.cam.ac.uk/Research/HVG/HOL/documentation/description.dvi.gz>.
- [15] *The HOL System. Tutorial*. For HOL Kananaskis-2, March 8, 2004. Available on-line at <http://www.cl.cam.ac.uk/Research/HVG/HOL/documentation/tutorial.dvi.gz>.
- [16] S. Romanenko, C. Russo, P. Sestoft. *Moscow ML Library Documentation*. June 2000. Available on-line at <http://www.dina.kvl.dk/~sestoft/mosml/mosmllib.pdf>.
- [17] S. Romanenko, C. Russo, P. Sestoft. *Moscow ML Owner's Manual*. June 2000. Available on-line at <http://www.dina.kvl.dk/~sestoft/mosml/manual.pdf>.
- [18] S. Romanenko, C. Russo, P. Sestoft. *Moscow ML Language Overview*. June 2000. Available on-line at from <http://www.dina.kvl.dk/~sestoft/mosml/mosmlref.pdf>.
- [19] X. Fu, T. Bultan, and J. Su. *WSAT: A Tool for Formal Analysis of Web Services*, 16th International Conference on Computer Aided Verification, July 2004
- [20] Giovanna Di Marzo Serugendo and Nicolas Guelfi. *Formal development of java based web parallel applications*. In Proceedings of the Hawaii International Conference on System Sciences, 1998, 1998. Also available as Technical Report EPFL-DI No 97/248.
- [21] Svend Frolund and Kannan Govindarajan. *cl: A language for formally defining web services interactions*. Technical Report HPL-2003-208, Hewlett Packard Laboratories, October 2003.
- [22] May Haydar. *Formal framework for automated analysis and verification of web-based applications*. In 19th IEEE International Conference on Automated Software Engineering (ASE), pages 410413. IEEE Computer Society, 2004.
- [23] John Hughes. *Functional Pearls. Global variables in Haskell*
- [24] J. Peterson, A. Reid *Adding Records to Haskell* Department of Computer Science, Yale University, October 1998
- [25] Mark P. Jones, Simon Peyton Jones *Lightweight Extensible Records in Haskell*

- [26] T. Harris, Simon Marlow, Simon Peyton Jones, M. Herlihy. *Composable Memory Transactions* Microsoft Research
- [27] Simon Peyton Jones, A. Gordon, S. Finne. *Concurrent Haskell*
- [28] P. Thiemann *A Typed Representation for HTML and XML Documents in Haskell*
- [29] P. Thiemann *WASHWeb Authoring System in Haskell. User Manual*. August 2004
- [30] Koen Claessen, John Hughes *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*
- [31] Malcolm Wallace, Colin Runciman. *Haskell and XML: Generic Combinators or Type-Based Translation?*